Performant TCP for Low-Power Wireless Networks

Sam Kumar, Michael P Andersen, Hyung-Sin Kim, and David E. Culler,
University of California, Berkeley

https://www.usenix.org/conference/nsdi20/presentation/kumar

This paper is included in the Proceedings of the
17th USENIX Symposium on Networked Systems Design
and Implementation (NSDI '20)
February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

Performant TCP for Low-Power Wireless Networks

Sam Kumar, Michael P Andersen, Hyung-Sin Kim, and David E. Culler
University of California, Berkeley

Abstract

Low-power and lossy networks (LLNs) enable diverse applications integrating many resource-constrained embedded devices, often requiring interconnectivity with existing TCP/IP networks as part of the Internet of Things. But TCP has received little attention in LLNs due to concerns about its overhead and performance, leading to LLN-specific protocols that require specialized gateways for interoperability. We present a systematic study of a well-designed TCP stack in IEEE 802.15.4-based LLNs, based on the TCP protocol logic in FreeBSD. Through careful implementation and extensive experiments, we show that modern low-power sensor platforms are capable of running full-scale TCP and that TCP, counter to common belief, performs well despite the lossy nature of LLNs. By carefully studying the interaction between the transport and link layers, we identify subtle but important modifications to both, achieving TCP goodput within 25% of an upper bound (5–40x higher than prior results) and low-power operation commensurate to CoAP in a practical LLN application scenario. This suggests that a TCP-based transport layer, seamlessly interoperable with existing TCP/IP networks, is viable and performant in LLNs.

1 Introduction

Research on wireless networks of low-power, resource constrained, embedded devices—low-power and lossy networks (LLNs) in IETF terms [128]—blossomed in the late 1990s. To obtain freedom to tackle the unique challenges of LLNs, researchers initially departed from the established conventions of the Internet architecture [50, 68]. As the field matured, however, researchers found ways to address these challenges *within* the Internet architecture [70]. Since then, it has become commonplace to use IPv6 in LLNs via the 6LoWPAN [105] adaptation layer. IPv6-based routing protocols, like RPL [33], and application-layer transports over UDP, like CoAP [35], have become standards in LLNs. Most wireless sensor network (WSN) operating systems, such as TinyOS [95], RIOT [24], and Contiki [45], ship with IP implementations enabled and configured. Major industry vendors offer branded and supported 6LoWPAN stacks (e.g., TI SimpleLink, Atmel SmartConnect). A consortium, Thread [64], has formed around 6LoWPAN-based interoperability.

Despite these developments, transport in LLNs has remained ad-hoc and TCP has received little serious consideration. Many embedded IP stacks (e.g., OpenThread [106]) do not even support TCP, and those that do implement only a subset of its features (Appendix B). The conventional wisdom is that IP holds merit, but *TCP is ill-suited to LLNs*. This view is represented by concerns about TCP, such as:

- "TCP is not light weight ... and may not be suitable for implementation in low-cost sensor nodes with limited processing, memory and energy resources." [110] (Similar argument in [42], [75].)
- That "TCP is a connection-oriented protocol" is a poor match for WSNs, "where actual data might be only in the order of a few bytes." [114] (Similar argument in [110].)
- "TCP uses a single packet drop to infer that the network is congested." This "can result in extremely poor transport performance because wireless links tend to exhibit relatively high packet loss rates." [109] (Similar argument in [43], [44], [75].)

Such viewpoints have led to a plethora of WSN-specialized protocols and systems [110, 117, 132] for reliable data transport, such as PSFQ [130], STCP [75], RCRT [109], Flush [88], RMST [125], Wisden [138], CRRT [10], and CoAP [31], and for unreliable data transport, like CODA [131], ESRT [118], Fusion [71], CentRoute [126], Surge [94], and RBC [142].

As LLNs become part of the emerging Internet of Things (IoT), it behooves us to re-examine the transport question, with attention to how the landscape has shifted: (1) As part of IoT, LLNs must be interoperable with traditional TCP/IP networks; to this end, using TCP in LLNs simplifies IoT gateway design. (2) Popular IoT application protocols, like MQTT [39] and ZeroMQ [8], assume that TCP is used at the transport layer. (3) Some IoT application scenarios demand high link utilization and reliability on low-bandwidth lossy links. Embedded hardware has also evolved substantially, prompting us to revisit TCP's overhead. In this context, **this paper seeks to determine: Do the "common wisdom" concerns about TCP hold in a modern IEEE 802.15.4-based LLN? Is TCP (still) unsuitable for use in LLNs?**

To answer this question, we leverage the fully-featured TCP implementation in the FreeBSD Operating System (rather than a limited locally-developed implementation) and refactor it to work with the Berkeley Low-Power IP Stack (BLIP), Generic Network Stack (GNRC), and OpenThread network stack, on two modern LLN platforms (§5). Naïvely running TCP in an LLN indeed results in poor performance. However, upon close examination, we discover that this is not caused by the expected reasons, such as those listed above. The *actual* reasons for poor TCP performance include (1) small link-layer frames that increase TCP header overhead, (2) hidden terminal effects over multiple wireless hops, and (3) poor interaction between TCP and a duty-cycled link. Through

Challenge	Technique	Observed Improvement
Resource Constraints	Zero-Copy Send	Send Buffer: 50% less mem.
	In-Place Reass.	Recv Buffer: 38% less mem.
Link-Layer Properties	Large MSS	TCP Goodput: 4–5x higher
	Link Retry Delay	TCP Seg. Loss: 6% → 1%
Energy Constraints	Adaptive DC	HTTP Latency: $\approx$ 2x lower
	L2 Queue Mgmt.	TCP Radio DC: 3% → 2%

Table 1: Impact of techniques to run full-scale TCP in LLNs

a systematic study of TCP in LLNs, we develop techniques to resolve these issues (Table 1), uncover why the generally assumed problems do not apply to TCP in LLNs, and show that TCP perfoms well in LLNs once these issues are resolved:

We find that **full-scale TCP fits well within the CPU and memory constraints of modern LLN platforms** (§5, §6). Owing to the low bandwidth of a low-power wireless link, a small window size ($\approx$ 2 KiB) is sufficient to fill the bandwidth-delay product and achieve good TCP performance. This translates into small send/receive buffers that fit comfortably within the memory of modern WSN hardware. Furthermore, we propose using an atypical Maximum Segment Size (MSS) to manage header overhead and packet fragmentation. As a result, **full-scale TCP operates well in LLNs, with 5–40 times higher throughput than existing (relatively simplistic) embedded TCP stacks** (§6).

Hidden terminals are a serious problem when running TCP over multiple wireless hops. We propose adding a delay d between link-layer retransmissions, and demonstrate that it effectively reduces hidden-terminal-induced packet loss for TCP. We find that, because a small window size is sufficient for good performance in LLNs, **TCP is quite resilient to spurious packet losses, as the congestion window can recover to a full window quickly after loss** (§7).

To run TCP in a low-power context, we *adaptively* duty-cycle the radio to avoid poor interactions with TCP's self-clocking behavior. We also propose careful link-layer queue management to make TCP more robust to interference. We demonstrate that **TCP can operate at low power, comparable to alternatives tailored specifically for WSNs**, and that **TCP brings value for real IoT sensor applications** (§8).

We conclude that TCP is entirely capable of running on IEEE 802.15.4 networks and low-cost embedded devices in LLN application scenarios (§9). Since our improvements to TCP and the link layer maintain seamless interoperability with other TCP/IP networks, we believe that a TCP-based transport architecture for LLNs could yield considerable benefit.

In summary, this paper's contributions are:

1. We implement a full-scale TCP stack for low-power embedded devices and reduce its resource usage.
2. We identify the actual issues causing poor TCP performance and develop techniques to address them.
3. We explain why the expected insurmountable reasons for poor TCP performance actually do not apply.
4. We demonstrate that, once these issues are resolved, TCP performs comparably to LoWPAN-specialized protocols.

Table 1 lists our techniques to run TCP in an LLN. Although prior LLN work has already used various forms of link-layer delays [136] and adaptive duty-cycling [140], our work shows, where applicable, how to adapt these techniques to work well with TCP, and demonstrates that they can address the challenges of LLNs within a *TCP-based* transport architecture.

2 Background and Related Work

Since the introduction of TCP, a vast literature has emerged, focusing on improving it as the Internet evolved. Some representative areas include congestion control [9, 51, 62, 76], performance on wireless links [15, 27], performance in high-bandwidth environments [11, 30, 53, 65, 78], mobility [124], and multipath operation [115]. Below, we discuss TCP in the context of LLNs and embedded devices.

2.1 Low-Power and Lossy Networks (LLNs)

Although the term *LLN* can be applied to a variety of technologies, including LoRa and Bluetooth Low Energy, we restrict our attention in this paper to **embedded networks using IEEE 802.15.4**. Such networks are called LoWPANs [93]—Low-Power Wireless Personal Area Networks—in contrast to WANs, LANs (802.3), and WLANs (802.11). Outside of LoWPANs, TCP has been successfully adapted to a variety of networks, including serial [77], Wi-Fi [27], cellular [25, 100], and satellite [15,25] links. While an 802.15.4 radio can in principle be added as a NIC to any device, we consider only *embedded* devices where it is the primary means of communication, running operating systems like TinyOS [68], RIOT [24], Contiki [45], or FreeRTOS. These devices are currently built around microcontrollers with Cortex-M CPUs, which lack MMUs. Below, we explain how LoWPANs are different from other networks where TCP has been successfully adapted.

Resource Constraints. When TCP was initially adopted by ARPANET in the early 1980s, contemporary Internet citizens—typically minicomputers and high-end workstations, but not yet personal computers—usually had at least 1 MiB of RAM. 1 MiB is tiny by today's standards, yet the LLN-class devices we consider in this work have *1-2 orders of magnitude less RAM than even the earliest computers connected with TCP/IP*. Due to energy constraints, particularly SRAM leakage, RAM size in low-power MCUs does not follow Moore's Law. For example, comparing Hamilton [83], which we use in this work, to TelosB [113], an LLN platform from 2004, shows only a 3.2x increase in RAM size over 16 years. This has caused LLN-class embedded devices to have a different balance of resources than conventional systems, a trend that is likely to continue well into the future. For example, whereas conventional computers have historically had roughly 1 MiB of RAM for every MIPS of CPU, as captured by the 3M rule, Hamilton has $\approx$ 50 DMIPS of CPU but only 32 KiB of RAM.

Link-Layer Properties. IEEE 802.15.4 is a low-bandwidth, wireless link with an MTU of only 104 bytes. The research community has explored using TCP with links that are *separately* low-bandwidth, wireless [27], or low-MTU [77], but addressing these issues *together* raises new challenges. For example, RTS-CTS, used in WLANs to avoid hidden terminals, has high overhead in LoWPANs [71, 136] due to the small MTU—control frames are comparable in size to data frames. Thus, LoWPAN researchers have moved away from RTS-CTS, instead carefully designing application traffic patterns to avoid hidden terminals [71, 88, 112]. Unlike Wi-Fi/LTE, LoWPANs do not use physical-layer techniques like adaptive modulation/coding or multi-antenna beamforming. Thus, they are directly impacted by link quality degradation due to varying environmental conditions [112, 127]. Additionally, IEEE 802.15.4 coexists with Wi-Fi in the 2.4 GHz frequency band, making Wi-Fi interference particularly relevant in indoor settings [99]. As LoWPANs are *embedded* networks, there is no human in the loop to react to and repair bad link quality.

Energy Constraints. Embedded nodes—the "hosts" of an LLN—are subject to strict power constraints. Low-power radios consume almost as much energy listening for a packet as they do when actually sending or receiving [20, 83]. Therefore, it is customary to *duty-cycle* the radio, keeping it in a low-power sleep state, in which it cannot send or receive data, most of the time [70, 112, 139]. The radio is only *occasionally* turned on to send/receive packets or determine if reception is likely. This requires *Media Management Control (MMC)* protocols [70, 112, 139] at the link layer to ensure that frames destined for a node are delivered to it only when its radio is on and listening. Similarly, the CPU also consumes a significant amount of energy [83], and must be kept idle most of the time.

Over the past 20 years, LLN researchers have addressed these challenges, but only in the context of special-purpose networks highly tailored to the particular application task at hand. The remaining open question is how to do so with a general-purpose reliable transport protocol like TCP.

2.2 TCP/IP for Embedded LLN-Class Devices

In the late 1990s and early 2000s, developers attempted to bring TCP/IP to embedded and resource-constrained systems to connect them to the Internet, usually over serial or Ethernet. Such systems [32, 80] were often designed with a specific application—often, a web server—in mind. These TCP/IP stacks were tailored to the specific applications at hand and were not suitable for general use. uIP ("micro IP") [42], introduced in 2002, was a standalone *general* TCP/IP stack optimized for 8-bit microcontrollers and serial or Ethernet links. To minimize resource consumption to run on such platforms, uIP omits standard features of TCP; for example, it allows only a single outstanding (unACKed) TCP segment per connection, rather than a sliding window of in-flight data.

Since the introduction of uIP, embedded networks have changed substantially. With *wireless* sensor networks and IEEE 802.15.4, various low-power networking protocols have been developed to overcome lossy links with strict energy and resource constraints, from S-MAC [139], B-MAC [112], X-MAC [34], and A-MAC [49], to Trickle [96] and CTP [59]. Researchers have viewed TCP as unsuitable, however, questioning end-to-end recovery, loss-triggered congestion control, and bi-directional data flow in LLNs [44]. Furthermore, WSNs of this era typically did not even use IP; instead, each WSN was designed specifically to support a particular application [89, 102, 138]. Those that require global connectivity rely on application-specific "base stations" or "gateways" connected to a TCP/IP network, treating the LLN like a peripheral interconnect (e.g., USB, bluetooth) rather than a network in its own right. This is because the prevailing sentiment at the time was that LLNs are too different from other types of networks and have to operate in too extreme conditions for the layered Internet architecture to be appropriate [50].

In 2007, the 6LoWPAN adaptation layer [105] was introduced, enabling IPv6 over IEEE 802.15.4. IPv6 has since been adopted in LLNs, bringing forth IoT [70]. uIP has been ported to LLNs [48], and IPv6 routing protocols, like RPL [33], and UDP-based application-layer transports, like CoAP [35], have emerged in LLNs. Representative operating systems, like TinyOS and Contiki, implement UDP/RPL/IPv6/6LoWPAN network stacks with IEEE 802.15.4-compatible MMC protocols for 16-bit platforms like TelosB [113].

TCP, however, is not widely adopted in LLNs. The few LLN studies that use TCP [47, 60, 67, 70, 72, 86, 144] generally use a simplified TCP stack (Appendix B), such as uIP.

In summary, despite the acceptance of IPv6, LLNs remain highly tailored at the transport layer to the application at hand. They typically use application-specific protocols on top of UDP; of such protocols, CoAP [31] has the widest adoption. In this context, this paper explores whether adopting TCP—and more broadly, the ecosystem of IP-based protocols, rather than IP alone—might bring value to LLNs moving forward.

3 Motivation: The Case for TCP in LLNs

As explained in §2, LLN design has historically been highly tailored to the specific application task at hand, for maximum efficiency. For example, PSFQ broadcasts data from a single source node to all others, RMST supports "directed diffusion" [73], and CoAP is tied to REST semantics. But embedded networks are not just isolated devices (e.g., peripheral interconnects like USB or bluetooth)—they are now true Internet citizens, and should be designed as such.

In particular, the recent megatrend of IoT requires LLNs to have a greater degree of *interoperability* with regular TCP/IP networks. Yet, LLN-specific protocols lack a clear separation between the transport and application layers, requiring *application-layer gateways* to communicate with TCP/IP-based services. This has encouraged IoT applications to develop as vertically-integrated silos, where devices cooperate only within an individual application or a particular manufacturer's ecosystem, with little to no interoperability *between* applications or with the general TCP/IP-based Internet. This phenomenon, sometimes called the "CompuServe of Things," is a serious obstacle to the IoT vision [57,97,104,133,141]. In contrast, other networks are seamlessly interoperable with the rest of the Internet. Accessing a new web application from a laptop does not require any new functionality at the Wi-Fi access point, but running a new application in a gateway-based LLN *does* require additional application-specific functionality to be installed at the gateway.

In this context, TCP-enabled LLN devices would be first-class citizens of the Internet, natively interoperable with the rest of the Internet via TCP/IP. They could use IoT protocols that assume a TCP-based transport layer (e.g., MQTT [39]) and security tools for TCP/IP networks (e.g., stateful firewalls), without an application-layer gateway. In addition, while traditional LLN applications like environment monitoring can be supported by unreliable UDP, certain applications do require high throughput and reliable delivery (e.g., anemometry (Appendix D), vibration monitoring [81]). TCP, *if it performs well in LLNs*, could benefit these applications.

Adopting TCP in LLNs may also open an interesting research agenda for IoT. TCP is the default transport protocol outside of LLNs, and history has shown that, to justify other transport protocols, application characteristics must offer substantial opportunity for optimization (e.g., [55,134,135]). If TCP becomes a viable option in LLNs, it would raise the bar for application-specific LLN protocols, resulting in some potentially interesting alternatives.

Although adopting TCP in LLNs could yield significant benefit and an interesting agenda, its feasibility and performance remain in question. This motivates our study.

4 Empirical Methodology

This section presents our methodology, carefully chosen to ground our study of full-scale TCP in LLNs.

4.1 Network Stack

Transport layer. That only a few full-scale TCP stacks exist, with a body of literature covering decades of refining, demonstrates that developing a feature-complete implementation of TCP is complex and error-prone [111]. Using a well-tested TCP implementation would ensure that results from our measurement study are due to the TCP *protocol*, not an artifact of the TCP *implementation* we used. Thus, we leverage the TCP implementation in FreeBSD 10.3 [56] to ground our study. We ported it to run in embedded operating systems and resource-constrained embedded devices (§4.2).

To verify the effectiveness of full-scale TCP in LLNs, we compare with CoAP [123], CoCoA [29], and unreliable UDP. CoAP is a standard LLN protocol that provides reliability on top of UDP. It is the most promising LLN alternative to TCP, gaining momentum in both academia [29,38,90,119,121,129] and industry [3,79], with adoption by Cisco [5,41], Nest/Google [4], and Arm [1,2]. CoCoA [29] is a recent proposal that augments CoAP with RTT estimation.

It is attractive to compare TCP to a variety of commercial systems, as has been done by a number of studies in LTE/WLANs [55,135]. Unfortunately, multihop LLNs have not yet reached the level of maturity to support a variety of commercial offerings; only CoAP has an appreciable level of commercial adoption. Other protocols are research proposals that often (1) are implemented for now-outdated operating systems and hardware or exist only in simulation [10,75,88], (2) target a very specific application paradigm [125,130,138], and/or (3) do not use IP [75,88,109,130]. We choose CoAP and CoCoA because they are not subject to these constraints.

	TelosB	Hamilton	Firestorm	Raspberry Pi
CPU	MSP430	Cortex-M0+	Cortex-M4	Cortex-A53
RAM	10 KiB	32 KiB	64 KiB	256 MB
ROM	48 KiB	256 KiB	512 KiB	SD Card

Table 2: Comparison of the platforms we used (Hamilton and Firestorm) to TelosB and Raspberry Pi

Layers 1 to 3. Because it is burdensome to place a border router with LAN connectivity within wireless range of every low-power host (e.g., sensor node), it is common to transmit data (e.g., readings) over *multiple* wireless LLN hops. Although each sensor must be battery-powered, it is reasonable to have a wall-powered LLN router node within transmission range of it.[1] This motivates Thread[2] [64,87], a recently developed protocol standard that constructs a multihop LLN over IEEE 802.15.4 links with *wall-powered, always-on* router nodes and *battery-powered, duty-cycled* leaf nodes. We use OpenThread [106], an open-source implementation of Thread.

Thread decouples routing from energy efficiency, providing a full-mesh topology among routers, frequent route updates, and asymmetric bidirectional routing for reliability. Each *leaf node* duty cycles its radio, and simply chooses a core router with good link quality, called its *parent*, as its next hop to all other nodes. The duty cycling uses *listen-after-send* [120]. A leaf node's parent stores downstream packets destined for that leaf node, until the leaf node sends it a *data request* message. A leaf node, therefore, can keep its radio powered off most of the time; infrequently, it sends a data request message to its parent, and turns on its radio for a short interval afterward to listen for downstream packets queued at its parent. Leaf nodes may send upstream traffic at any time. Each node uses CSMA-CA for medium access.

4.2 Embedded Hardware

We use two embedded hardware platforms: Hamilton [83] and Firestorm [18]. Hamilton uses a SAMR21 SoC with a 48 MHz Cortex-M0+, 256 KiB of ROM, and 32 KiB of RAM. Firestorm uses a SAM4L 48 MHz Cortex-M4 with 512 KiB of ROM and 64 KiB of RAM. While these platforms are more powerful than the TelosB [113], an older LLN platform widely

[1] The assumption of powered "core routers" is reasonable for most IoT use cases, which are typically indoors. Recent IoT protocols, such as Thread [64] and BLEmesh [63], take advantage of powered core routers.

[2] Thread has a large amount of industry support with a consortium already consisting of over 100 members [6], and is used in real IoT products sold by Nest/Google [7]. Given this trend, using Thread makes our work timely.

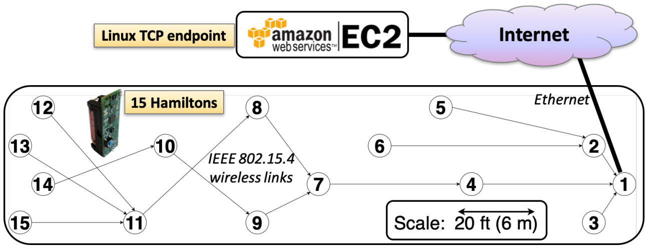


Figure 1: Snapshot of uplink routes in OpenThread topology at transmission power of -8 dBm (5 hops). Node 1 is the border router with Internet connectivity.

used in past studies, they are heavily resource-constrained compared to a Raspberry Pi (Table 2). Both platforms use the AT86RF233 radio, which supports IEEE 802.15.4. We use its standard data rate, 250 kb/s. We use Hamilton/OpenThread in our experiments; for comparison, we provide some results from Firestorm and other network stacks in Appendix A.

Handling automatic radio features. The AT86RF233 radio has built-in hardware support for link-layer retransmissions and CSMA-CA. However, it automatically enters low-power mode during CSMA backoff, during which it does not listen for incoming frames [20]. This behavior, which we call *deaf listening*, interacts poorly with TCP when radios are always on, because TCP requires bidirectional flow of packets—data in one direction and ACKs in the other. This may initially seem concerning, as deaf listening is an important power-saving feature. Fortunately, this problem disappears when using OpenThread's listen-after-send duty-cycling protocol, as leaf nodes never transmit data when listening for downstream packets. For experiments with always-on radios, we do not use the radio's capability for hardware CSMA and link retries; instead, we perform these operations in software.

Multihop Testbed. We construct an indoor LLN testbed, depicted in Figure 1, with 15 Hamiltons where node 1 is configured as the border router. OpenThread forms a 3-to-5-hop topology at transmission power of -8 dBm. Embedded TCP endpoints (Hamiltons) communicate with a Linux TCP endpoint (server on Amazon EC2) via the border router. During working hours, interference is present in the channel, due to people in the space using Wi-Fi and Bluetooth devices in the 2.4 GHz frequency band. At night, when there are few/no people in the space, there is much less interference.

5 Implementation of *TCPlp*

We seek to answer the following two questions: (1) Does full-scale TCP fit within the limited memory of modern LLN platforms? (2) How can we integrate a TCP implementation from a traditional OS into an embedded OS? To this end, we develop a TCP stack for LLNs based on the TCP implementation in FreeBSD 10.3, called *TCPlp* [91], on multiple embedded operating systems, RIOT OS [24] and TinyOS [95]. We use *TCPlp* in our measurement study in future sections.

Although we carefully preserved the protocol logic in the FreeBSD TCP implementation, achieving correct and performant operation on sensor platforms was a nontrivial effort. We had to modify the FreeBSD implementation according to the concurrency model of each embedded network stack and the timer abstractions provided by each embedded operating system (Appendix A). Our other modifications to FreeBSD, aimed at reducing memory footprint, are described below.

	Protocol	Socket Layer	`posix_sockets`
ROM	19972 B	6216 B	5468 B
RAM (Active)	364 B	88 B	48 B
RAM (Passive)	12 B	88 B	48 B

Table 3: Memory usage of *TCPlp* on RIOT OS. We also include RIOT's `posix_sockets` module, used by *TCPlp* to provide a Unix-like interface.

5.1 Connection State for *TCPlp*

As discussed in Appendix B, *TCPlp* includes features from FreeBSD that improve standard communication, like a sliding window, New Reno congestion control, zero-window probes, delayed ACKs, selective ACKs, TCP timestamps, and header prediction. *TCPlp*, however, omits some features in FreeBSD's TCP/IP stack. We omit dynamic window scaling, as buffers large enough to necessitate it ($\geq$ 64 KiB) would not fit in memory. We omit the urgent pointer, as it not recommended for use [61] and would only complicate buffering. Certain security features, such as host cache, TCP signatures, SYN cache, and SYN cookies are outside the scope of this work. We do, however, retain challenge ACKs [116].

We use separate structures for *active sockets* used to send and receive bytes, and *passive sockets* used to listen for incoming connections, as passive sockets require less memory.

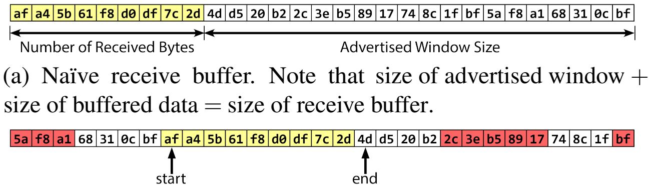
Table 3 depicts the memory footprint of *TCPlp* on RIOT OS. The memory required for the protocol and application state of an active TCP socket fits in a few hundred bytes, less than 1% of the available RAM on the Cortex-M4 (Firestorm) and 2% of that on the Cortex-M0+ (Hamilton). Although *TCPlp* includes heavyweight features not traditionally included in embedded TCP stacks, it fits well within available memory.

5.2 Memory-Efficient Data Buffering

Existing embedded TCP stacks, such as uIP and BLIP, allow *only one TCP packet in the air*, eschewing careful implementation of send and receive buffers [86]. These buffers, however, are key to supporting TCP's sliding window functionality. We observe in §6.2 that *TCPlp* performs well with only 2-3 KiB send and receive buffers, which comfortably fit in memory even when naïvely pre-allocated at compile time. Given that buffers dominate *TCPlp*'s memory usage, however, we discuss techniques to optimize their memory usage.

5.2.1 Send Buffer: Zero-Copy

Zero-copy techniques [28, 40, 82, 98, 101] were devised for situations where the time for the CPU to copy memory is a significant bottleneck. Our situation is very different; the radio, not the CPU, is the bottleneck, owing to the low bandwidth of IEEE 802.15.4. By using a zero-copy send buffer,


af a4 5b 61 f8 d0 df 7c 2d 4d d5 20 b2 2c 3e b5 89 17 74 8c 1f bf 5a f8 a1 68 31 0c bf
Number of Received Bytes
Advertised Window Size
5a f8 a1 68 31 0c bf af a4 5b 61 f8 d0 df 7c 2d 4d d5 20 b2 2c 3e b5 89 17 74 8c 1f bf
start
end

(a) Naïve receive buffer. Note that size of advertised window + size of buffered data = size of receive buffer.

(b) Receive buffer with in-place reassembly queue. In-sequence data (yellow) is kept in a circular buffer, and out-of-order segments (red) are written in the space past the received data.

Figure 2: Naïve and final TCP receive buffers

however, we can avoid allocating memory to intermediate buffers that would otherwise be needed to copy data, thereby reducing the network stack's total memory usage.

In TinyOS, for example, the BLIP network stack supports vectored I/O; an outgoing packet passed to the IPv6 layer is specified as an `iovec`. Instead of allocating memory in the packet heap for each outgoing packet, *TCPlp* simply creates `iovecs` that point to existing data in the send buffer. This decreases the required size of the packet heap.

Unfortunately, zero-copy optimizations were not possible for the OpenThread implementation, because OpenThread does not support vectored I/O for sending packets. The result is that the *TCPlp* implementation requires a few kilobytes of additional memory for the send buffer on this platform.

5.2.2 Receive Buffer: In-Place Reassembly Queue

Not all zero-copy optimizations are useful in the embedded setting. In FreeBSD, received packets are passed to the TCP implementation as `mbufs` [137]. The receive buffer and reassembly buffer are `mbuf` chains, so data need not be copied out of `mbufs` to add them to either buffer or recover from out-of-order delivery. Furthermore, buffer sizes are chosen dynamically [122], and are merely a *limit* on their actual size. In our memory-constrained setting, such a design is dangerous because its memory usage is nondeterministic; there is additional memory overhead, due to headers, if the data are delivered in many small packets instead of a few large ones.

We opted for a flat array-based circular buffer for the receive buffer in *TCPlp*, primarily owing to its determinism in a limited-memory environment: buffer space is reserved at *compile-time*. Head/tail pointers delimit which part of the array stores in-sequence data. To reduce memory consumption, we store out-of-order data in the same receive buffer, at the same position as if they were in-sequence. We use a bitmap, not head/tail pointers, to record where out-of-order data are stored, because out-of-order data need not be contiguous. We call this an *in-place reassembly queue* (Figure 2).

6 TCP in a Low-Power Network

In this section, we characterize how full-scale TCP interacts with a low-power network stack, resource-constrained hardware, and a low-bandwidth link.

6.1 Reducing Header Overhead using MSS

In traditional networks, it is customary to set the Maximum Segment Size (MSS) to the link MTU (or path MTU) minus the size of the TCP/IP headers. IEEE 802.15.4 frames, however, are *an order of magnitude smaller* than frames in traditional networks (Table 4). The TCP/IP headers consume more than half of the frame's available MTU. As a result, TCP performs poorly, incurring more than 50% header overhead.

	Fast Ethernet	Wi-Fi	Ethernet	802.15.4
Capacity	100 Mb/s	54 Mb/s	10 Mb/s	250 kb/s
MTU	1500 B	1500 B	1500 B	104–116 B
Tx Time	0.12 ms	0.22 ms	1.2 ms	4.1 ms

Table 4: Comparison of TCP/IP links

Header	802.15.4	6LoWPAN	IPv6	TCP	Total
1st Frame	11–23 B	5 B	2–28 B	20–44 B	38–107 B
*n*th Frame	11–23 B	5–12 B	0 B	0 B	16–35 B

Table 5: Header overhead with 6LoWPAN fragmentation

Earlier approaches to running TCP over low-MTU links (e.g., low-speed serial links) have used TCP/IP header compression based on per-flow state [77] to reduce header overhead. In contrast, the 6LoWPAN adaptation layer [105], designed for LLNs, supports only *flow-independent* compression of the IPv6 header using shared link-layer state, a clear departure from per-flow techniques. A key reason for this is that the compressor and decompressor in an LLN (host and border router) are separated by several IP hops[3], making it desirable for intermediate nodes to be able to determine a packet's IP header without per-flow context (see §10 of [105]).

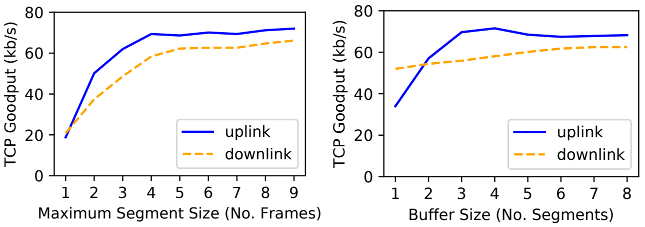
That said, compressing TCP headers separately from IP addresses using per-flow state is a promising approach to further amortize header overhead. There is preliminary work in this direction [22, 23], but it is based on uIP, which has one in-flight segment, and does not fully specify how to resynchronize compression state after packet loss with a multi-segment window. It is also not officially standardized by the IETF.

Therefore, this paper takes an approach orthogonal to header compression. We instead choose an MSS larger than the link MTU admits, *relying on fragmentation at the lower layers to decrease header overhead*. Fragmentation is handled by 6LoWPAN, which acts at Layer 2.5, between the link and network layers. Unlike end-to-end IP fragmentation, the 6LoWPAN fragments exist only within the LLN, and are reassembled into IPv6 packets when leaving the network.

Relying on fragmentation is effective because, as shown in Table 5, TCP/IP headers consume space in the first fragment, but not in subsequent fragments. Using an excessively large MSS, however, decreases reliability because the loss of one fragment results in the loss of an entire packet. Existing work [21] has identified this trade-off and investigated it in simulation in the context of power consumption. We investigate it in the context of goodput in a live network.

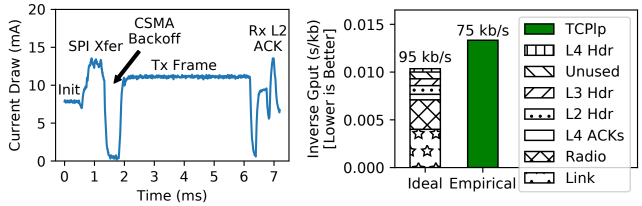
Figure 3a shows the bandwidth as the MSS varies. As

[3] Thread deliberately does not abstract the mesh as a single IP link. Instead, it organizes the LLN mesh as a set of *overlapping link-local scopes*, using IP-layer routing to determine the path packets take through the mesh [70].



(a) Effect of varying MSS (b) Effect of varying buffer size

Figure 3: TCP goodput over one IEEE 802.15.4 hop



(a) Unicast of a single frame, measured with an oscilloscope (b) *TCPlp* goodput compared with raw link bandwidth and overheads

Figure 4: Analysis of overhead limiting *TCPlp*'s goodput

expected, we see poor performance at a small MSS due to header overhead. Performance gains diminish when the MSS becomes larger than 5 frames. We recommend using an MSS of about 5 frames, but it is reasonable to decrease it to 3 frames if more wireless loss is expected. **Despite the small frame size of IEEE 802.15.4, we can effectively amortize header overhead for TCP using an atypical MSS.** Adjusting the MSS is orthogonal to TCP header compression. We hope that widespread use of TCP over 6LoWPAN, perhaps based on our work, will cause TCP header compression to be separately investigated and possibly used together with a large MSS.

6.2 Impact of Buffer Size

Whereas simple TCP stacks, like uIP, allow only one in-flight segment, full-scale TCP requires complex buffering (§5.2). In this section, we vary the size of the buffers (send buffer for uplink experiments and receive buffer for downlink experiments) to study how it affects the bandwidth. In varying the buffer size, we are directly affecting the size of TCP's flow window. We expect throughput to increase with the flow window size, with diminishing returns once it exceeds the bandwidth-delay product (BDP). The result is shown in Figure 3b. **Goodput levels off at a buffer size of 3 to 4 segments (1386 B to 1848 B), indicating that the buffer size needed to fill the BDP fits comfortably in memory.** Indeed, the BDP in this case is about $125\text{kb/s} \cdot 0.1\text{s} \approx 1.6\text{KiB}$.[4]

Downlink goodput at a buffer size of one segment is unusually high. This is because FreeBSD does not delay ACKs if the receive buffer is full, reducing the effective RTT from ≈ 130 ms to ≈ 70 ms. Indeed, goodput is very sensitive to RTT when the buffer size is small, because TCP exhibits "stop-and-wait" behavior due to the small flow window.

6.3 Upper Bound on Single-Hop Goodput

We consider TCP goodput between two nodes over the IEEE 802.15.4 link, over a single hop without any border router. Using the Hamilton/OpenThread platform, we are able to achieve 75 kb/s.[5] Figure 4b lists various sources of overhead that limit *TCPlp*'s goodput, along with the ideal upper bounds that they admit. **Link** overhead refers to the 250 kb/s link capacity. **Radio** overhead includes SPI transfer to/from the radio (i.e., packet copying [107]), CSMA, and link-layer ACKs, which cannot be pipelined because the AT86RF233 radio has only one frame buffer. A full-sized 127-byte frame spends 4.1 ms in the air at 250 kb/s, but the radio takes 7.2 ms to send it (Figure 4a), almost halving the link bandwidth available to a single node. This is consistent with prior results [107]. **Unused** refers to unused space in link frames due to inefficiencies in the 6LoWPAN implementation. Overall, we estimate a 95 kb/s upper bound on goodput (100 kb/s without TCP headers). Our 75 kb/s measurement is within 25% of this upper bound, substantially higher than prior work (Table 6). The difference from the upper bound is likely due to network stack processing and other real-world inefficiencies.

7 TCP Over Multiple Wireless Hops

We instrument TCP connections between Hamilton nodes in our multi-hop testbed, without using the EC2 server.

7.1 Mitigating Hidden Terminals in LLNs

Prior work over traditional WLANs has shown that hidden terminals degrade TCP performance over multiple wireless hops [58]. Using RTS/CTS for hidden terminal avoidance has been shown to be effective in WLANs. This technique has an unacceptably high overhead in LLNs [136], however, because data frames are small (Table 4), comparable in size to the additional control frames required. Prior work in LLNs has carefully designed application traffic, using rate control [71, 88] and link-layer delays [136], to avoid hidden terminals.

But prior work does not explore these techniques in the context of TCP. Unlike protocols like CoAP and simplified TCP implementations like uIP, a full-scale TCP flow has a *multi-segment sliding window* of unacknowledged data, making it unclear *a priori* whether existing LLN techniques will be sufficient. In particular, rate control seems sufficient because of bi-directional packet flow in TCP (data in one direction and ACKs in the other). So, rather than applying rate control, we attempt to avoid hidden terminals by adding a delay d between link-layer retries in addition to CSMA backoff. After a failed link transmission, a node waits for a random duration between 0 and d, before retransmitting the frame. The idea is

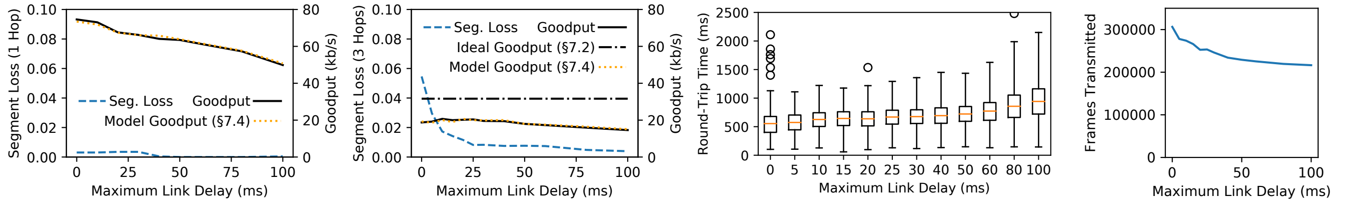
[4] We estimate the bandwidth as 125 kb/s rather than 250 kb/s to account for the radio overhead identified in §6.3.

[5] Appendix A.4 provides the corresponding goodput figures for Hamilton/GNRC and Firestorm/BLIP platforms, for comparison.

[6] One study [47] achieves ≈ 16 kb/s over multiple hops using the Linux TCP stack. We do not include it in Table 6 because it does not capture the resource constraints of LLNs—it uses traditional computers (PCs) for the end hosts—and does not consider hidden terminals—each hop uses a different wireless channel. It also uses TCP as a workload to evaluate a new link-layer protocol (burst forwarding), instead of evaluating TCP in its own right

	[144]	[22]	[67]	[86]	[69, 70]	This Paper (Hamilton Platform)
TCP Stack	uIP	uIP	uIP	BLIP	Arch Rock	*TCPlp* (RIOT OS, OpenThread)
Max. Seg Size	1 Frame	1 Frame	4 Frames	1 Frame	1024 bytes	5 Frames
Window Size	1 Seg.	1 Seg.	1 Seg.	1 Seg.	1 Seg.	1848 bytes (4 Seg.)
Goodput (One Hop)	1.5 kb/s	≈ 6.4 kb/s	≈ 12 kb/s	≈ 4.8 kb/s	15 kb/s	75 kb/s
Goodput (Multi-Hop)	≈ 0.55 kb/s	≈ 1.9 kb/s	≈ 12 kb/s	≈ 2.4 kb/s	9.6 kb/s	20 kb/s

Table 6: Comparison of *TCPlp* to existing TCP implementations used in network studies over IEEE 802.15.4 networks.[6] Goodput figures obtained by reading graphs in the original paper (rather than stated numbers) are marked with the ≈ symbol.



(a) TCP goodput, one hop (b) TCP goodput, three hops (c) RTT, three hops (outliers omitted) (d) Total frames sent, three hops

Figure 5: Effect of varying time between link-layer retransmissions. Reported "segment loss" is the loss rate of TCP segments, not individual IEEE 802.15.4 frames. It includes only losses not masked by link-layer retries.

that if two frames collide due to a hidden terminal, the delay will prevent their link-layer retransmissions from colliding.

We modified OpenThread, which previously had no delay between link retries, to implement this. As expected, single-hop performance (Figure 5a) decreases as the delay between link retries increases; hidden terminals are not an issue in that setting. Packet loss is high for the multihop experiment (Figure 5b) when the link retry delay is 0, as is expected from hidden terminals. **Adding a small delay between link retries, however, effectively reduces packet loss.** Making the delay too large raises the RTT (Figure 5c).

We prefer a smaller frame/segment loss rate, even if goodput stays the same, in order to make more efficient use of network resources. Therefore, we prefer a moderate delay ($d = 40$ ms) to a small delay ($d = 5$ ms), even though both provide the same goodput, because the frame and segment loss rates are smaller when d is large (Figures 5b and 5d).

7.2 Upper Bound on Multi-Hop Goodput

Comparing Figures 5a and 5b, goodput over three wireless hops is substantially smaller than goodput over a single hop. Prior work has observed similar throughput reductions over multiple hops [86, 107]. It is due to radio scheduling constraints inherent in the multihop setting, which we describe in this section. Let B be the bandwidth over a single hop.

Consider a two-hop setup: $S \to R_1 \to D$. R_1 cannot receive a frame from S while sending a frame to D, because its radio cannot transmit and receive simultaneously. Thus, the maximum achievable bandwidth over two hops is $\frac{B}{2}$.

Now consider a three-hop setup: $S \to R_1 \to R_2 \to D$. By the same argument, if a frame is being transferred over $R_1 \to R_2$, then neither $S \to R_1$ nor $R_2 \to D$ can be active. Furthermore, if a frame is being transferred over $R_2 \to D$, then R_1 can hear that frame. Therefore, $S \to R_1$ cannot transfer a frame at that time; if it does, then its frame will collide at R_1 with the frame being transferred over $R_2 \to D$. Thus, the maximum bandwidth is $\frac{B}{3}$. We depict this ideal upper bound in Figure 5b, taking B to be the ideal single-hop goodput from §6.3.

In setups with more than three hops, every set of three adjacent hops is subject to this constraint. The first hop and fourth hop, however, may be able to transfer frames simultaneously. Therefore, the maximum bandwidth is still $\frac{B}{3}$. In practice, goodput may fall slightly because transmissions from a node may *interfere* with nodes multiple hops away, even if they can only be received by its immediate neighbors.

We made empirical measurements with $d = 40$ ms to validate this analysis. Goodput over one hop was 64.1 kb/s; over two hops, 28.3 kb/s; over three hops, 19.5 kb/s; and over four hops, 17.5 kb/s. This roughly fits the model.

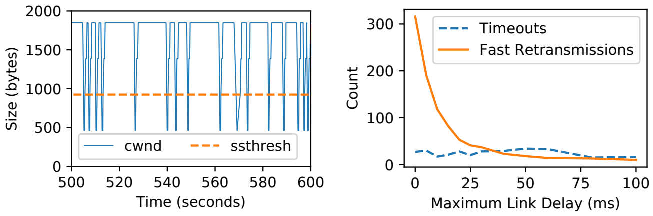
This analysis justifies why the same window size works well for both the one-hop experiments and the three-hop experiments in §7.1. Although the RTT is three times higher, the bandwidth-delay product is approximately the same. **Crucially, this means that the 2 KiB buffer size we determined in §6.2, which fits comfortably in memory, remains applicable for up to three wireless hops.**

7.3 TCP Congestion Control in LLNs

Recall that small send/receive buffers of only 1848 bytes (4 TCP segments) each are enough to achieve good TCP performance. This profoundly impacts TCP's congestion control mechanism. For example, consider Figure 5b. It is remarkable that throughput is almost the same at $d = 0$ ms and $d = 30$ ms, despite having 6% packet loss in the first case and less than 1% packet loss in the second.

Figure 6a depicts the congestion window over a 100 second interval during the $d = 0$ ms experiment.[7] Interestingly,

[7] All congestion events in Figure 6a were fast retransmissions, except for one timeout at $t = 569$ s. cwnd is temporarily set to 1 MSS during fast retransmissions due to an artifact of FreeBSD's implementation of SACK recovery. For clarity, we cap cwnd at the size of the send buffer, and we remove fluctuations in cwnd which resulted from "bad retransmissions" that the FreeBSD implementation corrected in the course of its normal execution.



(a) TCP cwnd for $d = 0$, three hops (b) TCP loss recovery, three hops

Figure 6: Congestion behavior of TCP over IEEE 802.15.4

the cwnd graph is far from the canonical sawtooth shape (e.g., Figure 11(b) in [26]); cwnd is almost always maxed out even though losses are frequent (6%). This is specific to small buffers. In traditional environments, where links have higher throughput and buffers are large, it takes longer for cwnd to recover after packet loss, greatly limiting the sending rate with frequent packet losses. In contrast, **in LLNs, where send/receive buffers are small, cwnd recovers to the maximum size quickly after packet loss, making TCP performance robust to packet loss.**

Congestion behavior also provides insight into loss patterns, as shown in Figure 6b. Fast retransmissions (used for isolated losses) become less frequent as d increases, suggesting that they are primarily caused by hidden-terminal-related losses. Timeouts do not become less frequent as d is increased, suggesting that they are caused by something else.

7.4 Modeling TCP Goodput in an LLN

Our findings in §7.3 suggest that, in LLNs, cwnd is limited by the buffer size, not packet loss. To validate this, we analytically model TCP performance according to our observations in §7.3, and then check if the resulting model is consistent with the data. Comprehensive models of TCP, which take window size limitations into account, already exist [108]; in contrast, our model is *intentionally simple* to provide intuition.

Observations in §7.3 suggest that we can neglect the time it takes the congestion window to recover after packet loss. So, we model a TCP connection as *binary*: either it is sending data with a full window, or it is not sending new data because it is recovering from packet loss. According to this model, a TCP flow alternates between *bursts* when it is transmitting at a full window, and *rests* when it is in recovery and not sending new data. Burst lengths depend on the packet loss rate p and rest lengths depend on RTT. This approach leads to the following model (full derivation is in Appendix C):

$$B = \frac{\text{MSS}}{\text{RTT}} \cdot \frac{1}{\frac{1}{w} + 2p} \qquad (1)$$

where B, the TCP goodput, is written in terms of the maximum segment size MSS, round-trip time RTT, packet loss rate p $(0 < p < 1)$, and window size w (sized to BDP, in packets). Figures 5a and 5b include the predicted goodput as dotted lines, calculated according to Equation 1 using the empirical RTT and segment loss rate for each experiment. **Our model of TCP goodput closely matches the empirical results.**

An established model of TCP outside of LLNs is [92, 103]:

$$B = \frac{\text{MSS}}{\text{RTT}} \cdot \sqrt{\frac{3}{2p}} \qquad (2)$$

Equation 2 fundamentally relies on there being many competing flows, so we do not expect it to match our empirical results from §7.3. But, given that existing work examining TCP in LLNs makes use of this formula to ground new algorithms [72], the differences between Equations 1 and 2 are interesting to study. In particular, Equation 1 has an added $\frac{1}{w}$ in the denominator and depends on p rather than $\sqrt{p}$, explaining, mathematically, how TCP in LLNs is more robust to small amounts of packet loss. We hope Equation 1, together with Equation 4 in Appendix C, will provide a foundation for future research on TCP in LLNs.

8 TCP in LLN Applications

To demonstrate that TCP is practical for real IoT use cases, we compare its performance to that of CoAP, CoCoA, and unreliable UDP in three workloads inspired by real application scenarios: web server, sense-and-send, and event detection. We evaluate the protocols over multiple hops with duty-cycled radios and wireless interference, present in our testbed in the day (§4.2). In our experiments, nodes 12–15 (Figure 1) send data to a server running on Amazon EC2. The RTT from the border router to the server was ≈ 12 ms, much smaller than within the low-power mesh (≈ 100-300 ms).

In our preliminary experiments, we found that in the presence of simultaneous TCP flows, tail drops at a relay node significantly impacted fairness. Implementing Random Early Detection (RED) [54] with Explicit Congestion Notification (ECN) support solved this problem. Therefore, we use RED and ECN for experiments in this section with multiple flows. While such solutions have sometimes been problematic since they are implemented in routers, they are more natural in LLNs because the intermediate "routers" relaying packets in an LLN typically also participate in the network as hosts.

We generally use a smaller MSS (3 frames) in this section, because it is more robust to interference in the day (§6). We briefly discuss how this affects our model in Appendix C, but leave a rigorous treatment to future work.

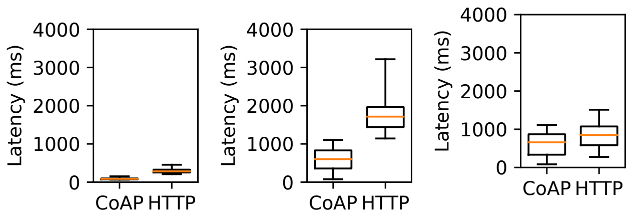
Running TCP in these application scenarios motivates **Adaptive Duty Cycle** and **Finer-Grained Link Queue Management**, which we introduce below as they are needed.

8.1 Web Server Application Scenario

To study TCP with multiple wireless hops and duty cycling, we begin with a web server hosted on a low-power device. We compare HTTP/TCP and CoAP/UDP (§4.1).

8.1.1 Latency Analysis

An HTTP request requires two round-trips: one to establish a TCP connection, and another for request/response. CoAP requires only one round trip (no connection establishment) and has smaller headers. Therefore, CoAP has a lower latency



(a) No duty cycling (b) 1 s sleep interval (c) 1 s sleep interval with adaptive duty cycle

Figure 7: Latency of web request: CoAP vs. HTTP/TCP

than HTTP/TCP when using an always-on link (Figure 7a). Even so, the latency of HTTP/TCP in this case is well below 1 second, not so large as to degrade user experience.

We now explore how a duty-cycled link affects the latency. Recall that leaf nodes in OpenThread (§4.1) periodically poll their parent to receive downstream packets, and keep their radios in a low-power sleep state between polls. We set the *sleep interval*—the time that a node waits between polls—to 1 s and show the latency in Figure 7b. Interestingly, HTTP's minimum observed latency is much higher than CoAP's, more than is explained by its additional round trip.

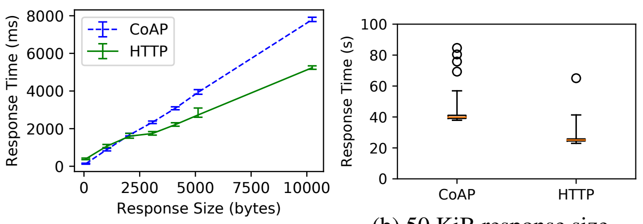
Upon investigation, we found that this is because **the self-clocking nature of TCP [76] interacts poorly with the duty-cycled link**. Concretely, the web server receives the SYN packet when it polls its parent, and sends the SYN-ACK immediately afterward, at the *beginning* of the next sleep interval. The web server therefore waits for the *entire* sleep interval before polling its parent again to receive the HTTP request, thereby experiencing the worst-case latency for the second round trip. We also observed this problem for batch transfer over TCP; TCP's self-clocking behavior causes it to consistently experience the worst-case round-trip time.

To solve this problem, we propose a technique called **Adaptive Duty Cycling**. After the web server receives a SYN, it *reduces the sleep interval* in anticipation of receiving an HTTP request. After serving the request, it restores the sleep interval to its old value. Unlike early LLN link-layer protocols like S-MAC [140] that use an adaptive duty cycle, we use *transport-layer state* to inform the duty cycle. Figure 7c shows the latency with adaptive duty cycling, where the sleep interval is temporarily reduced to 100 ms after connection establishment. **With adaptive duty-cycling, the latency overhead of HTTP compared to CoAP is small, despite larger headers and an extra round trip for connection establishment.**

Adaptive duty cycling is also useful in high-throughput scenarios, and in situations with persistent TCP connections. We apply adaptive duty cycling to one such scenario in §8.2.

8.1.2 Throughput Analysis

In §8.1.1, the size of the web server's response was 82 bytes, intentionally small to focus on latency. In a real application, however, the response may be large (e.g., it may contain a batch of sensor readings). In this section, we explore larger response sizes. We use a short sleep interval of 100 ms. This is realistic because, using adaptive duty cycling, the sleep interval may be longer when the node is idle, and reduced to 100 ms only when transferring the response.



(a) Response time vs. size (b) 50 KiB response size

Figure 8: Goodput: CoAP vs. HTTP/TCP

Figure 8a shows the total time from dispatching the request to receiving the full response, as we vary the size of the response. It plots the median time, with quartiles shown in error bars. HTTP takes longer than CoAP when the response size is small (consistent with Figure 7), but CoAP takes longer when the response size is larger. This indicates that while HTTP/TCP has a greater fixed-size overhead than CoAP (higher y-intercept), it transfers data at a higher throughput (lower slope). TCP achieves a higher throughput than CoAP because CoAP sends response segments one-at-a-time ("stop and wait"), whereas TCP allows multiple segments to be in flight simultaneously ("sliding window").

To quantify the difference in throughput, we compare TCP and CoAP when transferring 50 KiB of data in Figure 8b. **TCP achieves 40% higher throughput compared to CoAP, over multiple hops and a duty-cycled link.**

8.1.3 Power Consumption

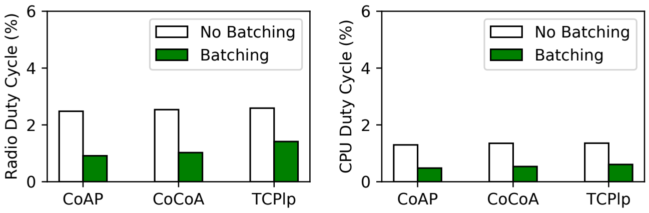
TCP consumes more energy than CoAP due to the extra round-trip at the beginning. In practice, however, a web server is interactive, and therefore will be *idle* most of the time. Thus, the idle power consumption dominates. For example, TCP keeps the radio on 35% longer than CoAP for a response size of 1024 bytes, but if the user makes one request every 100 seconds on average, this difference drops to only 0.35%.

Thus, we relegate in-depth power measurements to the sense-and-send application (§8.2), which is non-interactive.

8.2 Sense-and-Send Application Scenario

We turn our focus to the common *sense-and-send* paradigm, in which devices periodically collect sensor readings and send them upstream. For concreteness, we model our experiments on the deployment of anemometers in a building, a real-world LLN use case described in Appendix D. Anemometers collect measurements frequently (once per second), making heavy use of the transport protocol; given that our focus is on transport performance, this makes anemometers a good fit for our study. Other sensor deployments (e.g., temperature, humidity, building occupancy, etc.) sample data at a lower rate (e.g., 0.05 Hz), but are otherwise similar. Thus, *we expect our results to generalize to other sense-and-send applications.*

Nodes 12–15 (Figure 1) each generate one 82-byte reading every 1 second, and send it to the cloud server using either



(a) Radio duty cycle (b) CPU duty cycle

Figure 9: Effect of batching on power consumption

TCP or CoAP. We use most of the remaining RAM as an *application-layer queue* to prevent data from being lost if CoAP or TCP is in backoff after packet loss and cannot send out new data immediately. We make use of adaptive duty cycling for both TCP and CoAP, with a base sleep interval of four minutes (OpenThread's default) and decreasing it to 100 ms[8] when a TCP ACK or CoAP response is expected.

We measure a solution's *reliability* as the proportion of generated readings delivered to the server. Given that TCP and CoAP both guarantee reliability, a reliability measurement of less than 100% is caused by overflow of the application-layer queue due to poor network conditions preventing data from being reliably communicated as fast as they are generated. Generating data more slowly would result in higher reliability.

8.2.1 Performance in Favorable Conditions

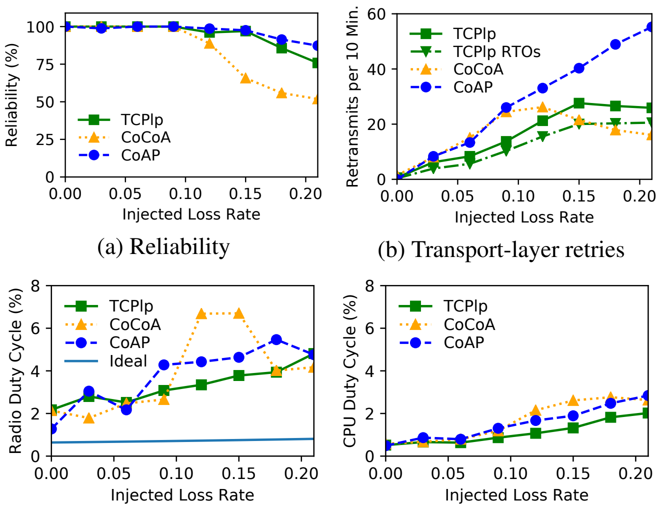
We begin with experiments in our testbed at night, when there is less wireless interference. We compare three setups: (1) CoAP, (2) CoCoA, and (3) *TCPlp*. We also compare two sending scenarios: (1) sending each sensor reading right away ("No Batching"), and (2) sending sensor readings in batches of 64 ("Batching") [89]. We ensure that packets in a CoAP batch are the same size as segments in TCP (five frames).

All setups achieved 100% reliability due to end-to-end acknowledgments (figures are omitted for brevity). Figures 9a and 9b also show that all the three protocols consume similar power; *TCP is comparable to LLN-specific solutions*.

Both the radio and CPU duty cycle are significantly smaller with batching than without batching. By sending data in batches, nodes can amortize the cost of sending data and waiting for a response. Thus, batching is the more realistic workload, so we use it to continue our evaluation.

8.2.2 Resilience to Packet Loss

In this section, we inject uniformly random packet loss at the border router and measure each solution.The result is shown in Figure 10. Note that the injected loss rate corresponds to the *packet-level* loss rate *after* link retries and 6LoWPAN reassembly. Although we plot loss rates up to 21%, *we consider loss rates* > *15% exceptional; we focus on the loss rate up to 15%*. A number of WSN studies have already achieved > 90% end-to-end packet delivery, using only link/routing layer techniques (not transport) [46, 84, 85]. In our testbed environment, we have not observed the loss rate exceed 15% for an extended time, even with wireless interference.



(a) Reliability (b) Transport-layer retries

(c) Radio duty cycle (d) CPU duty cycle

Figure 10: Performance with injected packet loss

Both CoAP and TCP achieve nearly 100% reliability at packet loss rates less than 15%, as shown in Figure 10a. At loss rates greater than 9%, CoCoA performs poorly. The reason is that CoCoA attempts to measure RTT for retransmitted packets, and conservatively calculates the RTT relative to the first transmission. This results in an inflated RTT value that causes CoCoA to delay longer before retransmitting, causing the application-layer queue to overflow. Full-scale TCP is immune to this problem despite measuring the RTT, because the TCP timestamp option allows TCP to unambiguously determine the RTT even for retransmitted segments.

Figures 10c and 10d show that, overall, **TCP and CoAP perform comparably in terms of radio and CPU duty cycle**. At 0% injected loss, *TCPlp* has a slightly higher duty cycle, consistent with Figure 9. At moderate packet loss, *TCPlp* appears to have a slightly lower duty cycle. This may be due to TCP's sliding window, which allows it to tolerate some ACK losses without retries. Additionally, Figure 10b shows that, although most of TCP's retransmissions are explained by timeouts, a significant portion were triggered in other ways (e.g., duplicate ACKs). In contrast, CoAP and CoCoA rely exclusively on timeouts, which has intrinsic limitations [143].

With exceptionally high packet loss rates (>15%), CoAP achieves higher reliability than TCP, because it "gives up" after just 4 retries; it exponentially increases the wait time between those retries, but then resets its RTO to 3 seconds when giving up and moving to the next packet. In contrast, TCP performs up to 12 retries with exponential backoff. Thus, TCP backs off further than CoAP upon consecutive packet losses, witnessed by the smaller retransmission count in Figure 10b, causing the application-layer queue to overflow more. This performance gap could be filled by parameter tuning.

We also consider an *ideal* "roofline" protocol to calculate a fairly loose lower bound on the duty cycle. This ideal protocol has the same header overhead as TCP, but learns which

[8] 100 ms is comparable to ContikiMAC's default sleep interval of 125 ms.

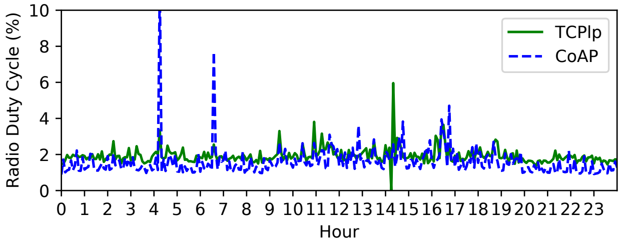


Figure 11: Radio duty cycle of TCP and CoAP in a lossy wireless environment, in one representative trial (losses are caused by natural human activity)

packets were lost for "free," without using ACKs or running MMC. Thus, it turns on its radio only to send out data and retransmit lost packets. The real protocols have much higher duty cycles than the ideal protocol would have (Figure 10c), suggesting that a significant amount of their overhead stems from determining which packets were lost—polling the parent node for downstream TCP ACKs/CoAP responses. This gap could be reduced by improving OpenThread's MMC protocol. For example, rather than using a fixed sleep interval of 100 ms when an ACK is expected, one could use exponential backoff to increase the sleep interval if an ACK is not quickly received. We leave exploring such ideas to future work.

8.2.3 Performance in Lossy Conditions

We compare the protocols over the course of a full day in our testbed, to study the impact of real wireless interference associated with human activity in an office. We focus on *TCPlp* and CoAP since they were the most promising protocols from the previous experiment. To ensure that *TCPlp* and CoAP are subject to similar interference patterns, we (1) run them simultaneously, and (2) hardcode adjacent *TCPlp* and CoAP nodes to have the same first hop in the multihop topology.

Improving Queue Management. OpenThread's queue management interacts poorly with TCP in the presence of interference. When a duty-cycled leaf node sends a data request message to its parent, it turns its radio on and listens until it receives a reply (called an "indirect message"). In OpenThread, the parent finishes sending its current frame (which may require link retries in the presence of interference), and then sends the indirect message. The duty-cycled leaf node keeps its radio on during this time, causing its radio duty cycle to increase. This is particularly bad for TCP, as its sliding window makes it more likely for the parent node to be in the middle of sending a frame when it receives a data request packet from a leaf node. Thus, **we modified OpenThread to allow indirect messages to preempt the current frame *in between link-layer retries***, to minimize the time that duty-cycled leaf nodes must wait for a reply with their radios on. Both TCP and CoAP benefitted from this; TCP benefitted more because it suffered more from the problem to begin with.

Power Consumption. To improve power consumption for both TCP and CoAP, we adjusted parameters according to the lossy environment: (1) we enabled link-layer retries for indirect messages, (2) we decreased the data request timeout and performed link-layer retries more rapidly for indirect messages, to deliver them to leaves more quickly, and (3) given the high level of daytime interference, we decreased the MSS from five frames to three frames (as in §8).

Protocol	Reliability	Radio DC	CPU DC
TCPlp	99.3%	2.29%	0.973%
CoAP	99.5%	1.84%	0.834%
Unrel., no batch	93.4%	1.13%	0.52%
Unrel., with batch	95.3%	0.734%	0.30%

Table 7: Performance in the testbed over a full day, averaged over multiple trials. The ideal protocol (§8.2.2) would have a radio DC of ≈ 0.63%–0.70% under similarly lossy conditions.

Figure 11 depicts the radio duty cycle of TCP and CoAP for a trial representative of our overall results. **CoAP maintains a lower duty cycle than *TCPlp* outside of working hours, when there is less interference; *TCPlp* has a slightly lower duty cycle than CoAP during working hours, when there is more wireless interference.** *TCPlp*'s better performance at a higher loss rate is consistent with our results from §8.2.2. At a lower packet loss rate, TCP performs slightly worse than CoAP. This could be due to hidden terminal losses; more retries, on average, are required for indirect messages for TCP, causing leaf nodes to stay awake longer. Overall, CoAP and *TCPlp* perform similarly (Table 7).

8.2.4 Unreliable UDP

As a point of comparison, we repeat the sense-and-send experiment using a UDP-based protocol that *does not provide reliability*. Concretely, we run CoAP in "nonconfirmable" mode, in which it does not use transport-layer ACKs or retransmissions. The result is in the last two rows of Table 7. Compared to unreliable UDP, reliable approaches increase the radio/CPU duty cycle by 3x, in exchange for nearly 100% reliability. That said, the corresponding decrease in battery life will be *less* than 3x, because other sources of power consumption (reading from sensors, idle current) are also significant.

For other sense-and-send applications that sample at a lower rate, TCP and CoAP would see higher reliability (less application queue loss), but UDP would not similarly benefit (no application queue). Furthermore, the power consumption of TCP, CoAP, and unreliable UDP would all be closer together, given that the radio and CPU spend more time idle.

8.3 Event Detection Application Scenario

Finally, we consider an application scenario where multiple flows compete for available bandwidth in an LLN. One such scenario is event detection: sensors wait until an interesting event occurs, at which point they report data upstream at a high data rate. Because such events tend to be correlated, multiple sensors send data simultaneously.

Nodes 12-15 in our testbed simultaneously transmit data to the EC2 instance (Figure 1), which measures the goodput

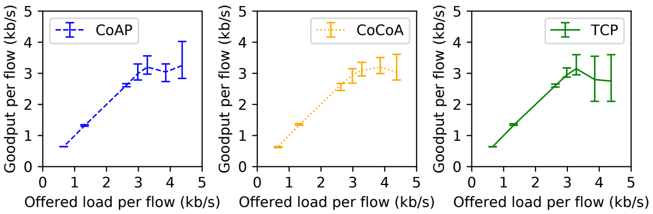


Figure 12: CoAP, CoCoA, and TCP with four competing flows

of each flow. We use the same duty-cycling policy as in §8.2. We divide each flow into 40-second intervals, measure the goodput in each interval, and compute the median and quartiles of goodput across all flows and intervals. The median gives a sense of aggregate goodput, and the quartiles gives a sense of fairness (quartiles close to the median are better).

Figure 12 shows the median and quartiles (as error bars) as the offered load increases. For small offered load, the per-flow goodput increases linearly. Once the aggregate load saturates the network, goodput declines slightly and the interquartile range increases, due to inefficiences in independent flows competing for bandwidth. **Overall, TCP performs similarly to CoAP and CoCoA, indicating that TCP's congestion control remains effective despite our observations in §7.3 that it behaves differently in LLNs.**

9 Conclusion

TCP is the *de facto* reliability protocol in the Internet. Over the past 40 years, new physical-, datalink-, and application-layer protocols have evolved alongside TCP, and supporting good TCP performance was a consideration in their design. TCP is the obvious performance baseline for new transport-layer proposals. To warrant adoption, novel transports must be *much* better than TCP in the intended application domain.

In contrast, when LLN research flourished two decades ago, LLN hardware could not run full-scale TCP. The original system architecture for networked sensors [68], for example, targeted an 8-bit MCU with only 512 *bytes* of memory. It naturally became taken for granted that TCP is too heavy for LLNs. Furthermore, contemporary research on TCP in WLANs [27] suggested that TCP would perform poorly in LLNs even if the resource constraints were surmounted.

In revisiting the TCP question, after the resource constraints relaxed, we find that the expected pitfalls of wireless TCP actually do not carry over to LLNs. Although naïve TCP indeed performs poorly in LLNs, this is not due to fundamental problems with TCP as were observed in WLANs. Rather, it is caused by incompatibilities with a low-power link layer, which likely arose because canonical LLN protocols were developed in the absence of TCP considerations. We show how to fix these incompatibilities while preserving seamless interoperability with other TCP/IP networks. This enables a viable TCP-based transport architecture for LLNs.

Our results have several implications for LLNs moving forward. First, **the use of lightweight protocols that emulate part of TCP's functionality, like CoAP, needs to be reconsidered.** Protocol stacks like OpenThread should support full-scale TCP as an option. TCP should also serve as a benchmark to assess new LLN transport proposals.

Second, **full-scale TCP will influence the design of networked systems using LLNs.** Such systems are presently designed with application-layer gateways in mind (§3). Using TCP/IP in the LLN itself would allow the use of commodity network management tools, like firewalls and NIDS. TCP would also allow the application-layer gateway to be replaced with a network-layer router, allowing clients to interact with LLN applications in much the same way as a Wi-Fi router allows users to interact with web applications. This is much more flexible than the status quo, where each LLN application needs application-specific functionality to be installed at the gateway [141]. In cases where a new LLN transport protocol is truly necessary, the new protocol may be wise to consider the byte-stream *abstraction* of TCP. This would allow the application-layer gateway to be replaced by a *transport-layer gateway*. The mere presence of a transport layer, distinct from the application layer, goes a long way to providing interoperability with the rest of the Internet.

Third, **UDP-based protocols will still have a place in LLNs, just as they have a place in the Internet.** UDP is used for applications that benefit from greater control of segment transmission and loss response than TCP provides. These are typically real-time or multimedia applications where losing information is preferable to late delivery. It is entirely seemly for some sensing applications in LLNs, particularly those with similar real-time constraints, to transfer data using UDP-based protocols, even if TCP is an option. But TCP *still* benefits such applications by providing a reliable channel for control information. For example, TCP may be used for device configuration, or to provide a shell for debugging, without yet another reliability protocol.

In summary, LLN-class devices are ready to become first-class citizens of the Internet. To this end, we believe that TCP should have a place in the LLN architecture moving forward, and that it will help put the "I" in IoT for LLN-class devices.

Acknowledgments

We thank the anonymous reviewers, including in prior submissions, and our shepherd, Keith Winstein, for their invaluable feedback. We are thankful to researchers in the BETS research group, including Kaifei Chen, Gabe Fierro, and Jack Kolb, for their feedback on early drafts of this paper, and to Prabal Dutta and Sylvia Ratnasamy for their advice and discussion. We also thank Albert Goto for his help with the LLN testbed.

This research is supported by the Department of Energy Grant DE-EE0007685, California Energy Commission, Intel Corporation, NSF Grant CPS-1239552, Fulbright Scholarship Program, UC Berkeley, and NSF Graduate Research Fellowship Program under Grant DGE-1752814. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Device management connect. https://www.arm.com/products/iot/pelion-iot-platform/device-management/connect. Accessed: 2018-09-09.

[2] Java speaks CoAP. https://community.arm.com/iot/b/blog/posts/java-speaks-coap. Accessed: 2018-09-09.

[3] MQTT and CoAP, IoT protocols. https://www.eclipse.org/community/eclipse_newsletter/2014/february/article2.php. Accessed: 2018-09-09.

[4] OpenThread. https://openthread.io/. Accessed: 2018-09-09.

[5] Software configuration guide, Cisco IOS release 15.2(5)ex (catalyst digital building series switches). https://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst_digital_building_series_switches/software/15-2_5_ex/configuration_guide/b_1525ex_consolidated_cdb_cg/b_1525ex_consolidated_cdb_cg_chapter_0111101.html. Accessed: 2018-09-09.

[6] Thread group. https://www.threadgroup.org/thread-group#OurMembers. Accessed: 2018-09-11.

[7] What is Thread. https://www.threadgroup.org/What-is-Thread#threadready. Accessed: 2018-09-12.

[8] ZeroMQ. http://zeromq.org/. Accessed: 2019-01-29.

[9] A. Afanasyev, N. Tilley, P. Reiher, and L. Kleinrock. Host-to-host congestion control for TCP. *IEEE Communications Surveys & Tutorials*, 12(3), 2010.

[10] M. M. Alam and C. S. Hong. CRRT: congestion-aware and rate-controlled reliable transport in wireless sensor networks. *IEICE Transactions on Communications*, 92(1), 2009.

[11] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *SIGCOMM*. ACM, 2010.

[12] M. Allman. TCP byte counting refinements. *ACM SIGCOMM Computer Communication Review*, 29(3), 1999.

[13] M. Allman. TCP congestion control with appropriate byte counting (ABC). RFC 3465, 2003.

[14] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing TCP's loss recovery using limited transmit. RFC 3042, 2000.

[15] M. Allman, D. Glover, and L. Sanchez. Enhancing TCP over satellite channels using standard mechanisms. RFC 2488, 1999.

[16] M. Allman and V. Paxson. On estimating end-to-end network path properties. *ACM SIGCOMM Computer Communication Review*, 29(4), 1999.

[17] M. Allman, V. Paxson, and E. Blanton. TCP congestion control. RFC 5681, 2009.

[18] M. P Andersen, G. Fierro, and D. E. Culler. System design for a synergistic, low power mote/BLE embedded platform. In *IPSN*. ACM/IEEE, 2016.

[19] E. Arens, A. Ghahramani, R. Przybyla, M. P Andersen, S. Min, T. Peffer, P. Raftery, M. Zhu, V. Luu, and H. Zhang. Measuring 3D indoor air velocity via an inexpensive low-power ultrasonic anemometer. *Energy and Buildings*, 211, 2020.

[20] Atmel Corporation. *Low Power, 2.4GHz Transceiver for ZigBee, RF4CE, IEEE 802.15.4, 6LoWPAN, and ISM Applications*, 2014. Preliminary Datasheet.

[21] A. Ayadi, P. Maillé, and D. Ros. TCP over low-power and lossy networks: tuning the segment size to minimize energy consumption. In *NTMS*. IEEE, 2011.

[22] A. Ayadi, P. Maillé, D. Ros, L. Toutain, and T. Zheng. Implementation and evaluation of a TCP header compression for 6LoWPAN. In *IWCMC*. IEEE, 2011.

[23] A. Ayadi, D. Ros, and L. Toutain. TCP header compression for 6LoWPAN: draft-aayadi-6lowpan-tcphc-01. Technical report, 2010. https://tools.ietf.org/id/draft-aayadi-6lowpan-tcphc-01.

[24] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch. RIOT: an open source operating system for low-end embedded devices in the IoT. *IEEE Internet of Things Journal*, 2018.

[25] H. Balakrishnan, V. N. Padmanabhan, and R. H. Katz. The effects of asymmetry on TCP performance. In *MobiCom*. ACM, 1997.

[26] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM Transactions on Networking*, 5(6), 1997.

[27] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz. Improving TCP/IP performance over wireless networks. In *MobiCom*. ACM, 1995.

[28] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. In *SOSP*. ACM, 1989.

[29] A. Betzler, C. Gomez, I. Demirkol, and J. Paradells. CoAP congestion control for the Internet of Things. *IEEE Communications Magazine*, 54(7), 2016.

[30] D. Borman, B. Braden, and V. Jacobson. TCP extensions for high performance. (7323), 2014.

[31] C. Bormann, A. P. Castellani, and Z. Shelby. CoAP: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2), 2012.

[32] G. Borriello and R. Want. Embedded computation meets the world wide web. *Communications of the ACM*, 43(5), 2000.

[33] A. Brandt, J. W. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J.-P. Vasseur, and R. Alexander. RPL: IPv6 routing protocol for low-power and lossy networks. RFC 6550, 2012.

[34] M. Buettner, G. V. Yee, E. Anderson, and R. Han. X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks. In *SenSys*. ACM, 2006.

[35] A. P. Castellani, M. Gheda, N. Bui, M. Rossi, and M. Zorzi. Web services for the Internet of Things through CoAP and EXI. In *ICC*. IEEE, 2011.

[36] D. D. Clark. The structuring of systems using upcalls. In *SOSP*. ACM, 1985.

[37] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications magazine*, 27(6), 1989.

[38] W. Colitti, K. Steenhaut, N. De Caro, B. Buta, and V. Dobrota. Evaluation of constrained application protocol for wireless sensor networks. In *LANMAN*. IEEE, 2011.

[39] MQTT Community. MQTT. http://mqtt.org. Accessed: January 25, 2018.

[40] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *SOSP*. ACM, 1993.

[41] P. Duffy. Beyond MQTT: A Cisco view on IoT protocols. https://blogs.cisco.com/digital/beyond-mqtt-a-cisco-view-on-iot-protocols. Accessed: 2018-09-09.

[42] A. Dunkels. Full TCP/IP for 8-bit architectures. In *MobiSys*. ACM, 2003.

[43] A. Dunkels, J. Alonso, and T. Voigt. Making TCP/IP viable for wireless sensor networks. *SICS Research Report*, 2003.

[44] A. Dunkels, J. Alonso, T. Voigt, H. Ritter, and J. Schiller. Connecting wireless sensornets with TCP/IP networks. In *International Conference on Wired/Wireless Internet Communications*. Springer, 2004.

[45] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN*. IEEE, 2004.

[46] S. Duquennoy, B. Al Nahas, O. Landsiedel, and T. Watteyne. Orchestra: Robust mesh networks through autonomously scheduled TSCH. In *SenSys*. ACM, 2015.

[47] S. Duquennoy, F. Österlind, and A. Dunkels. Lossy links, low power, high throughput. In *SenSys*. ACM, 2011.

[48] M. Durvy, J. Abeillé, P. Wetterwald, C. O'Flynn, B. Leverett, E. Gnoske, M. Vidales, G. Mulligan, N. Tsiftes, N. Finne, and A. Dunkels. Making sensor networks IPv6 ready. In *SenSys*. ACM, 2008.

[49] P. Dutta, S. Dawson-Haggerty, Y. Chen, C.-J. M. Liang, and A. Terzis. Design and evaluation of a versatile and efficient receiver-initiated link layer for low-power wireless. In *SenSys*. ACM, 2010.

[50] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *MobiCom*. ACM, 1999.

[51] K. Fall and S. Floyd. Simulation-based comparisons of tahoe, reno and SACK TCP. *ACM SIGCOMM Computer Communication Review*, 26(3), 1996.

[52] S. Floyd. TCP and explicit congestion notification. *ACM SIGCOMM Computer Communication Review*, 24(5), 1994.

[53] S. Floyd. HighSpeed TCP for large congestion windows. RFC 3649, 2003.

[54] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4), 1993.

[55] S. Fouladi, J. Emmons, E. Orbay, C. Wu, R. S. Wahby, and K. Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *NSDI*. USENIX, 2018.

[56] The FreeBSD Foundation. FreeBSD 10.3, 2016. https://www.freebsd.org/releases/10.3R/announce.html.

[57] J. Fürst, K. Chen, M. Aljarrah, and P. Bonnet. Leveraging physical locality to integrate smart appliances in non-residential buildings with ultrasound and bluetooth low energy. In *IoTDI*. IEEE, 2016.

[58] M. Gerla, K. Tang, and R. Bagrodia. TCP performance in wireless multi-hop networks. In *WMCSA*. IEEE, 1999.

[59] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. In *SenSys*. ACM, 2009.

[60] C. Gomez, A. Arcia-Moret, and J. Crowcroft. TCP in the Internet of Things: From ostracism to prominence. *IEEE Internet Computing*, 22(1), 2018.

[61] F. Gont and A. Yourtchenko. On the implementation of the TCP urgent mechanism. RFC 6093, 2011.

[62] L. A. Grieco and S. Mascolo. Performance evaluation and comparison of Westwood+, New Reno, and Vegas TCP congestion control. *ACM SIGCOMM Computer Communication Review*, 34(2), 2004.

[63] Bluetooth Mesh Working Group. Mesh profile v1.0, 2017.

[64] Thread Group. Thread, 2016. https://threadgroup.org.

[65] S. Ha, I. Rhee, and L. Xu. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review*, 42(5), 2008.

[66] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno modification to TCP's fast recovery algorithm. RFC 6582, 2012.

[67] K. Hewage, S. Duquennoy, V. Iyer, and T. Voigt. Enabling TCP in mobile cyber-physical systems. In *MASS*. IEEE, 2015.

[68] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS*. ACM, 2000.

[69] J. W. Hui. Personal Communication.

[70] J. W. Hui and D. E. Culler. IP is dead, long live IP for wireless sensor networks. In *SenSys*. ACM, 2008.

[71] B. Hull, K. Jamieson, and H. Balakrishnan. Mitigating congestion in wireless sensor networks. In *SenSys*. ACM, 2004.

[72] H. Im. *TCP Performance Enhancement in Wireless Networks*. PhD thesis, Seoul National University, 2015.

[73] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *MobiCom*. ACM, 2000.

[74] D. Italiano and A. Motin. Calloutng: a new infrastructure for timer facilities in the FreeBSD kernel. In *AsiaBSDCon*, 2013.

[75] Y. G. Iyer, S. Gandham, and S. Venkatesan. STCP: a generic transport layer protocol for wireless sensor networks. In *ICCCN*. IEEE, 2005.

[76] V. Jacobson. Congestion avoidance and control. In *SIGCOMM*. ACM, 1988.

[77] V. Jacobson. Compressing TCP/IP headers for low-speed serial links. RFC 1144, 1990.

[78] C. Jin, D. X. Wei, and S. H. Low. FAST TCP: motivation, architecture, algorithms, performance. In *INFOCOM*. IEEE, 2004.

[79] S. Johnson. Constrained application protocol: CoAP is IoT's 'modern' protocol. https://www.omaspecworks.org/constrained-application-protocol-coap-is-iots-modern-protocol/, https://internetofthingsagenda.techtarget.com/feature/Constrained-Application-Protocol-CoAP-is-IoTs-modern-protocol. Accessed: 2018-09-09.

[80] H.-T. Ju, M.-J. Choi, and J. W. Hong. An efficient and lightweight embedded web server for web-based network element management. *International Journal of Network Management*, 10(5), 2000.

[81] D. Jung, Z. Zhang, and M. Winslett. Vibration analysis for IoT enabled predictive maintenance. In *ICDE*. IEEE, 2017.

[82] Yousef A. Khalidi and Moti N. Thadani. An efficient zero-copy I/O framework for UNIX. Technical report, Mountain View, CA, USA, 1995.

[83] H.-S. Kim, M. P Andersen, K. Chen, S. Kumar, W. J. Zhao, K. Ma, and D. E. Culler. System architecture directions for post-SoC/32-bit networked sensors. In *SenSys*. ACM, 2018.

[84] H.-S. Kim, H. Cho, H. Kim, and S. Bahk. DT-RPL: Diverse bidirectional traffic delivery through RPL routing protocol in low power and lossy networks. *Computer Networks*, 126, 2017.

[85] H.-S. Kim, H. Cho, M.-S. Lee, J. Paek, J. Ko, and S. Bahk. MarketNet: An asymmetric transmission power-based wireless system for managing e-price tags in markets. In *SenSys*. ACM, 2015.

[86] H.-S. Kim, H. Im, M.-S. Lee, J. Paek, and S. Bahk. A measurement study of TCP over RPL in low-power and lossy networks. *Journal of Communications and Networks*, 17(6), 2015.

[87] H.-S. Kim, S. Kumar, and D. E. Culler. Thread/OpenThread: A compromise in low-power wireless multihop network architecture for the Internet of Things. *IEEE Communications Magazine*, 57(7), 2019.

[88] S. Kim, R. Fonseca, P. Dutta, A. Tavakoli, D. E. Culler, P. Levis, S. Shenker, and I. Stoica. Flush: A reliable bulk transport protocol for multihop wireless networks. In *SenSys*. ACM, 2007.

[89] S. Kim, S. Pakzad, D. E. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *IPSN*. ACM/IEEE, 2007.

[90] M. Kovatsch, M. Lanter, and Z. Shelby. Californium: Scalable cloud services for the Internet of Things with CoAP. In *IOT*. IEEE, 2014.

[91] S. Kumar, M. P Andersen, H.-S. Kim, and D. E. Culler. Bringing full-scale TCP to low-power networks. In *SenSys*. ACM, 2018.

[92] J. Kurose and K. Ross. *Computer Networking: A Top-Down Approach*, chapter 3, pages 278–279. 6th edition, 2013.

[93] N. Kushalnagar, G. Montenegro, and C. Schumacher. IPv6 over low-power wireless personal area networks (6LoWPANs): Overview, assumptions, problem statement, and goals. RFC 4919, 2007.

[94] P. Levis, N. Lee, M. Welsh, and D. E. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *SenSys*. ACM, 2003.

[95] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. E. Culler. *TinyOS: An operating system for sensor networks*. 2005.

[96] P. Levis, N. Patel, D. E. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI*. USENIX, 2004.

[97] A. A. Levy, J. Hong, L. Riliskis, P. Levis, and K. Winstein. Beetle: Flexible communication for bluetooth low energy. In *MobiSys*. ACM, 2016.

[98] Y.-C. Li and M.-L. Chiang. LyraNET: a zero-copy TCP/IP protocol stack for embedded operating systems. In *RTCSA*. IEEE, 2005.

[99] C.-J. M. Liang, N. B. Priyantha, J. Liu, and A. Terzis. Surviving Wi-Fi interference in low power ZigBee networks. In *SenSys*. ACM, 2010.

[100] R. Ludwig, A. Gurtov, and F. Khafizov. TCP over second (2.5G) and third (3G) generation wireless networks. RFC 3481, 2003.

[101] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *SOSP*. ACM, 1993.

[102] A. Mainwaring, D. E. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA*. ACM, 2002.

[103] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM SIGCOMM Computer Communication Review*, 27(3), 1997.

[104] A. McEwen. Risking a compuserve of things. https://mcqn.com/posts/wuthering-bytes-slides-risking-a-compuserve-of-things/. Accessed: 2018-12-08.

[105] G. Montenegro, N. Kushalnagar, J. W. Hui, and D. E. Culler. Transmission of IPv6 packets over IEEE 802.15.4 networks. RFC 4944, 2007.

[106] Google Nest. OpenThread, 2017. https://github.com/openthread/openthread.

[107] F. Österlind and A. Dunkels. Approaching the maximum 802.15.4 multi-hop throughput. In *HotEmNets*. ACM, 2008.

[108] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. *ACM SIGCOMM Computer Communication Review*, 28(4), 1998.

[109] J. Paek and R. Govindan. RCRT: Rate-controlled reliable transport for wireless sensor networks. In *SenSys*. ACM, 2007.

[110] Q. Pang, V. W. S. Wong, and V. C. M. Leung. Reliable data transport and congestion control in wireless sensor networks. *International Journal of Sensor Networks*, 3(1), 2008.

[111] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz. Known TCP implementation problems. RFC 2525, 1999.

[112] J. Polastre, J. Hill, and D. E. Culler. Versatile low power media access for wireless sensor networks. In *SenSys*. ACM, 2004.

[113] J. Polastre, R. Szewczyk, and D. E. Culler. Telos: Enabling ultra-low power wireless research. In *IPSN*. ACM/IEEE, 2005.

[114] M. A. Rahman, A. El Saddik, and W. Gueaieb. *Wireless Sensor Network Transport Layer: State of the Art*. 2008.

[115] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? Designing and implementing a deployable multipath TCP. In *NSDI*. USENIX, 2012.

[116] A. Ramaiah, M. Stewart, and M. Dalal. Improving TCP's robustness to blind in-window attacks. RFC 5961, 2010.

[117] A. J. D. Rathnayaka and V. M. Potdar. Wireless sensor network transport protocol: A critical review. *Journal of Network and Computer Applications*, 36(1), 2013.

[118] Y. Sankarasubramaniam, Ö. B. Akan, and I. F. Akyildiz. Esrt: event-to-sink reliable transport in wireless sensor networks. In *MobiHoc*. ACM, 2003.

[119] D. F. S. Santos, H. O. Almeida, and A. Perkusich. A personal connected health system for the Internet of Things based on the Constrained Application Protocol. *Computers & Electrical Engineering*, 44, 2015.

[120] T. Schmid, R. Shea, M. B. Srivastava, and P. Dutta. Disentangling wireless sensing from mesh networking. In *HotEmNets*, 2010.

[121] K. Seitz, S. Serth, K.-F. Krentz, and C. Meinel. Enabling en-route filtering for end-to-end encrypted CoAP messages. In *SenSys*. ACM, 2017.

[122] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP buffer tuning. In *SIGCOMM*. ACM, 1998.

[123] Z. Shelby, K. Hartke, and C. Bormann. The constrained application protocol (CoAP). RFC 7252, 2014.

[124] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *MobiCom*. ACM, 2000.

[125] F. Stann and J. Heidemann. RMST: Reliable data transport in sensor networks. In *SNPA*. IEEE, 2003.

[126] T. Stathopoulos, L. Girod, J. Heidemann, and D. Estrin. Mote herding for tiered wireless sensor networks. Technical Report 58, University of California, Los Angeles, Center for Embedded Networked Computing, December 2005.

[127] R. Szewczyk, J. Polastre, A. Mainwaring, and D. E. Culler. Lessons from a sensor network expedition. In H. Karl, A. Wolisz, and A. Willig, editors, *EWSN*. Springer Berlin Heidelberg, 2004.

[128] J.-P. Vasseur. Terms used in routing for low-power and lossy networks. RFC 7102, 2014.

[129] B. C. Villaverde, D. Pesch, R. De Paz Alberola, S. Fedor, and M. Boubekeur. Constrained Application Protocol for low power embedded networks: A survey. In *IMIS*. IEEE, 2012.

[130] C.-Y. Wan, A. T. Campbell, and L. Krishnamurthy. PSFQ: a reliable transport protocol for wireless sensor networks. In *WSNA*. ACM, 2002.

[131] C.-Y. Wan, S. B. Eisenman, and A. T. Campbell. CODA: Congestion detection and avoidance in sensor networks. In *SenSys*. ACM, 2003.

[132] C. Wang, K. Sohraby, Y. Hu, B. Li, and W. Tang. Issues of transport control protocols for wireless sensor networks. In *ICCCAS*. IEEE, 2005.

[133] P. Windley. The compuserve of things. http://www.windley.com/archives/2014/04/the_compuserve_of_things.shtml. Accessed: 2018-12-08.

[134] K. Winstein and H. Balakrishnan. Mosh: An interactive remote shell for mobile clients. In *ATC*. USENIX, 2012.

[135] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *NSDI*. USENIX, 2013.

[136] A. Woo and D. E. Culler. A transmission control scheme for media access in sensor networks. In *MobiCom*. ACM, 2001.

[137] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated*, volume 2, chapter 2. 1995.

[138] N. Xu, S. Rangwala, K. K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *SenSys*. ACM, 2004.

[139] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *INFOCOM*. IEEE, 2002.

[140] W. Ye, J. Heidemann, and D. Estrin. Medium access control with coordinated adaptive sleeping for wireless sensor networks. *IEEE/ACM Transactions on Networking*, 12(3), 2004.

[141] T. Zachariah, N. Klugman, B. Campbell, J. Adkins, N. Jackson, and P. Dutta. The Internet of Things has a gateway problem. In *HotMobile*. ACM, 2015.

[142] H. Zhang, A. Arora, Y.-R. Choi, and M. G. Gouda. Reliable bursty convergecast in wireless sensor networks. In *MobiHoc*. ACM, 2005.

[143] L. Zhang. Why TCP timers don't work well. *ACM SIGCOMM Computer Communication Review*, 16(3), 1986.

[144] T. Zheng, A. Ayadi, and X. Jiang. TCP over 6LoWPAN for industrial applications: An experimental study. In *NTMS*. IEEE, 2011.

A Impact of Network Stack Design

As mentioned in §5, we made nontrivial modifications to FreeBSD's TCP stack to port it to each embedded operating system and embedded network stack. Below we provide additional information about these changes, and about our implementations for platforms other than Hamilton/OpenThread.

A.1 Concurrency Model

GNRC and OpenThread (RIOT OS). RIOT OS provides threads as the basic unit of concurrency. Asynchronous interaction with hardware is done by interrupt handlers that preempt the current thread, perform a short operation in the interrupt context, and signal a related thread to perform any remaining operation outside of interrupt context. Then the thread is placed on the RIOT OS scheduler queue and is scheduled for execution depending on its priority.

The GNRC network stack for RIOT OS runs each network layer (or module) in a separate thread. Each thread has a priority and can be preempted by a thread with higher priority or by an interrupt. The thread for a lower network layer has higher priority than the thread for a higher layer.

The port of OpenThread for RIOT OS handles received packets in one thread and sends packets from another thread, where the thread for received packets has higher priority [83]. The rationale for this design is to ensure timely processing of received packets at the radio, which is especially important in the context of a high-throughput flow.

To adapt *TCPlp* for GNRC, we run the FreeBSD implementation as a single TCP-layer thread, whose priority is between that of the application-layer thread and the IPv6-layer thread. To adapt *TCPlp* for OpenThread on RIOT OS, we call the TCP protocol logic (`tcp_input()`) at the appropriate point along the receive path, and send packets from the TCP protocol logic (`tcp_output()`) using the established send path. As explained in Appendix A.2, we also use an additional thread for timer callbacks in RIOT OS.

Given that TCP state can be accessed concurrently from multiple threads—the TCP thread (GNRC) or receive thread (OpenThread), the application thread(s), and timer callbacks—we needed to synchronize access to it. The FreeBSD implementation allows fine-grained locking of connection state to allow different connections to be serviced in parallel on different CPUs. Given that low-power embedded sensors typically have only one CPU, however, we opted for simplicity, instead using a single global TCP lock for *TCPlp*.

BLIP (TinyOS). TinyOS uses an event-driven concurrency model based on split-phase operations, consisting of an event loop that executes on a *single* stack. For concurrency, TinyOS provides three types of unique operations: *commands* and *events*, which are executed immediately, and *tasks*, which are scheduled for execution after all preceding tasks are completed. An interrupt handler may preempt the current function, perform a short operation in the interrupt context using *asynchronous* events and commands, and *post* a task to perform any remaining computation later. To adapt the thread-based FreeBSD implementation to the event-driven TinyOS, we execute the primary functions of FreeBSD, such as `tcp_output()` and `tcp_input()`, within *tasks* outside of interrupt context. Because tasks in TinyOS cannot preempt each other, we remove the locking present in the FreeBSD TCP implementation.

A.2 Timer Event Management

Given that many TCP operations are based on timer events, achieving correct timer operation is important. For example, if an RTO timer event is dropped by the embedded operating system, the RTO timer will not be rescheduled, and the connection may hang.

For a simple and stable operation, many existing embedded TCP stacks, including the uIP, lwIP, and BLIP TCP stacks, rely on a periodic, fixed-interval clock in order to check for expired timeouts. Instead, *TCPlp* uses one-shot tickless timers as FreeBSD 10.3 does [74], which is beneficial in two ways: (1) When there are no scheduled timers, the tickless timers allow the CPU to sleep, rather than being needlessly woken up at a fixed interval, resulting in lower energy consumption [83]. (2) Unlike fixed periodic timers, which can only be serviced on the next tick after they expire, tickless timers can be serviced as soon as they expire. To obtain these advantages, however, an embedded operating system must robustly manage asynchronous timer callbacks.

TinyOS has a single event queue maintained by the scheduler. The semantics of TinyOS guarantee that a task can exist in the event queue only once, even if it is *posted* (i.e., scheduled for execution) multiple times before executing. Therefore, the event queue can be sized appropriately at compile-time to not overflow. Furthermore, TinyOS handles received packets in a separate queue than tasks. This ensures that TCP

	Protocol	Event Sched.	User Library
ROM	21352 B	1696 B	5384 B
RAM (Active)	488 B	40 B	36 B
RAM (Passive)	16 B	16 B	36 B

Table 8: Memory usage of *TCPlp* on TinyOS. Our implementation of *TCPlp* spans three modules: (1) protocol implementation, (2) event scheduler that injects callbacks into userspace, and (3) userland library.

timer callbacks will not be dropped.

This is not the case for RIOT OS. Timer callbacks either handle the timer entirely in interrupt context, or put an event on a thread's message queue, so that the thread performs the required callback operation. Each network protocol supported by RIOT OS has a single thread. Because a thread's message queue in RIOT OS is used to hold both received packets and timer events, there is no guarantee when a timer expires that there is enough space in the thread message queue to accept a timer event; if there is not enough space, RIOT OS drops the timer event. Furthermore, if a timer expires multiple times before its event is handled by the thread, multiple events for the same timer can exist simultaneously in the queue; *we cannot find an upper bound on the number of slots in the message queue used by a single timer*. To provide robust TCP operation on RIOT OS, we create a second thread used exclusively for TCP timers. We handle timers similarly to TinyOS' *post* operation, by preventing the message queue from having multiple callback events of a single timer. This eliminates the possibility of timer event drops.

A.3 Memory Usage: Connection State

To complement Table 3, which shows *TCPlp*'s memory footprint on RIOT OS, we include Table 8, which shows *TCPlp*'s memory footprint on TinyOS.

A.4 Performance Comparison

We consider TCP goodput between two embedded nodes over the IEEE 802.15.4 link, over a single hop without any border router, as we did in §6.3. We are able to produce a 63 kb/s goodput over a TCP connection between two Hamilton motes using RIOT's GNRC network stack. For comparison, we are able to achieve 71 kb/s using the BLIP stack on Firestorm, and 75 kb/s using the OpenThread network stack with RIOT OS on Hamilton. **This suggests that our results are reproducible across multiple platforms and embedded network stacks.** The minor performance degradation in GNRC is partially explained by its greater header overhead due to implementation differences, and by its IPC-based thread-per-layer concurrency architecture, which has known inefficiencies [36]. This suggests that the implementation of the underlying network stack, particularly with regard to concurrency, could affect TCP performance in LLNs.

	uIP	BLIP	GNRC	*TCPlp*
Flow Control	Yes	Yes	Yes	Yes
Congestion Control	N/A	No	Yes	Yes
RTT Estimation	Yes	No	Yes	Yes
MSS Option	Yes	No	Yes	Yes
OOO Reassembly	No	No	Yes	Yes
TCP Timestamps	No	No	No	Yes
Selective ACKs	No	No	No	Yes
Delayed ACKs	No	No	No	Yes

Table 9: Comparison of core features among embedded TCP stacks: uIP (Contiki), BLIP (TinyOS), GNRC (RIOT), and *TCPlp (this paper)*

B Comparison of Features in Embedded TCP Implementations

Table 9 compares the featureset of *TCPlp* to features in embedded TCP stacks. The TCP implementations in uIP and BLIP lack features core to TCP. uIP allows only one unACKed in-flight segment, eschewing TCP's sliding window. BLIP does not implement RTT estimation or congestion control. The TCP implementation in GNRC lacks features such as TCP timestamps, selective ACKs, and delayed ACKs, which are present in most full-scale TCP implementations.

Benefits of full-scale TCP. In addition to supporting the protocol-level features summarized in Table 9, *TCPlp* is likely more robust than other embedded TCP stacks because it is based on a well-tested TCP implementation. While seemingly minor, some details, implemented incorrectly by TCP stacks, have had important consequences for TCP's behavior [111]. *TCPlp* benefits from a thorough implementation of each aspect of TCP.

For example, *TCPlp*, by virtue of using the FreeBSD TCP implementation, benefits from a robust implementation of congestion control. *TCPlp* implements not only the basic New Reno algorithm, but also Explicit Congestion Notification [52], Appropriate Byte Counting [12, 13] and Limited Transmissions [14]. It also inherits from FreeBSD heuristics to identify and correct "bad retransmissions" (as in §2.8 of [16]): if, after a retransmission, the corresponding ACK is received very soon (within $\frac{\text{RTT}}{2}$ of the retransmission), the ACK is assumed to correspond to the originally transmitted segment as opposed to the retransmission. The FreeBSD implementation and *TCPlp* recover from such "bad retransmissions" by restoring `cwnd` and `ssthresh` to their former values before the packet loss. Aside from congestion control, *TCPlp* benefits from header prediction [37], which introduces a "fast code path" to process common-case TCP segments (in-sequence data and ACKs) more efficiently, and Challenge ACKs [116], which make it more difficult for an attacker to inject an RST into a TCP connection.

Enhancements such as these make us more confident that our observed results are fundamental to TCP, as opposed to artifacts of poor implementation. Furthermore, they allow us to focus on performance problems arising from the challenges of LLNs, as opposed to general TCP-related challenges that the research community has already solved in the context of traditional networks and operating systems.

C Derivation of TCP Model

This appendix provides the derivation of Equation 1, the model of TCP performance proposed in §7.4.

We think of a TCP flow as a sequence of bursts. A *burst* is a sequence of full windows of data successfully transferred, which ends in a packet loss. After this loss, the flow spends some time recovering from the packet loss, which we call a *rest*. Then, the next burst begins. Let w be the size of TCP's flow window, measured in segments (for our experiments in §7.3, we would have $w = 4$). Define b as the average number of windows sent in a burst. The goodput of TCP is the number of bytes sent in each burst, which is $w \cdot b \cdot \text{MSS}$, divided by the duration of each burst. A burst lasts for the time to transmit b windows of data, plus the time to recover from the packet loss that ended the burst. The time to transmit b windows is $b \cdot \text{RTT}$. We define t_{rec} to be the time to recover from the packet loss. Then we have

$$B = \frac{w \cdot b \cdot \text{MSS}}{b \cdot \text{RTT} + t_{\text{rec}}}. \tag{3}$$

The value of b depends on the packet loss rate. We define a new variable, p_{win}, which denotes the probability that at least one packet in a window is lost. Then $b = \frac{1}{p_{\text{win}}}$.

To complete the model, we must estimate t_{rec} and p_{win}.

The value of t_{rec} depends on whether the retransmission timer expires (called an RTO) or a fast retransmission is performed. If an RTO occurs, the total time lost is the excess time budgeted to the retransmit timer beyond one RTT, plus the time to retransmit the lost segments. We denote the time budgeted to the retransmit timer as ETO. So the total time lost due to a timeout, assuming it takes about 2 RTTs to recover lost segments, would be $(\text{ETO} - \text{RTT}) + 2 \cdot \text{RTT} = \text{ETO} + \text{RTT}$. After a fast retransmission, TCP enters a "fast recovery" state [17, 66]. Fast recovery requires buffer space to be effective, however. In particular, if the buffer contains only four TCP segments, then the lost packet, and three packets afterward which resulted in duplicate ACKs, account for the entire send buffer; therefore, TCP cannot send new data during fast recovery, and instead stalls for one RTT, until the ACK for the fast retransmission is received. In contrast, choosing a larger send buffer will allow fast recovery to more effectively mask this loss [122].

As discussed in §7.3, these two types of losses may be caused by different factors. Therefore, we do not attempt to distinguish them on basis of probability. Instead, we use a very simple model: $t_{\text{rec}} = \ell \cdot \text{RTT}$. The constant ℓ can be chosen to describe the number of "productive" RTTs lost due to a packet loss. Based on the estimates above, choosing $\ell = 2$ seems reasonable for our experiments in §7 which used a buffer size of four segments.

To model p_{win}, we assume that, in each window, segment losses are independent. This gives us $p_{\text{win}} = 1 - (1 - p)^w$, where p is the probability of an individual segment being lost (after link retries). Because p is likely to be small (less than 20%), we apply the approximation that $(1 - x)^a \approx 1 - ax$ for small x. This gives us $p_{\text{win}} \approx wp$.

Applying these equations for t_{rec} and p_{win}, along with some minor algebraic manipulation to put our equation in a similar form to Equation 2, we obtain our model for TCP performance in LLNs, for small w and p:

$$B = \frac{\text{MSS}}{\text{RTT}} \cdot \frac{1}{\frac{1}{w} + \ell p} \tag{4}$$

Equation 1, stated in §7.4, takes $\ell = 2$, as discussed above.

Generalizing the model. In §8, we generally use a smaller MSS (3 frames) than we used in §7. Furthermore, duty-cycling increases the RTT. It is natural to ask whether our conclusions in §7, on which the model is based, still hold in this setting. With a sleep interval of 100 ms, we qualitatively observed that, although `cwnd` tends to recover more slowly after loss, due to the smaller MSS and larger RTT, it is still "maxed out" past the BDP most of the time. Therefore, we expect our conclusion, that TCP is more resilient to packet loss, to also apply in this setting.

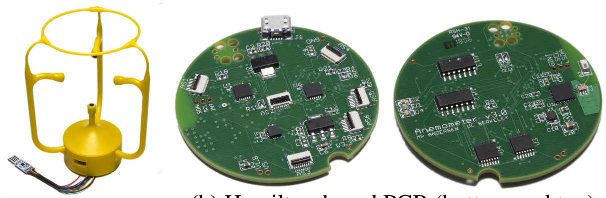
One may consider adapting our model for this setting by choosing a larger value of ℓ to reflect the fact that `cwnd` recovers from loss less quickly due to the smaller MSS. It is possible, however, that one could derive a better model by explicitly modeling the phase when `cwnd` is recovering, similar to other existing TCP models (in contrast to our model above, where we assume that the TCP flow is binary—either transmitting at a full window, or in backoff after loss). We leave exploration of this idea to future work.

D Anemometry: An LLN Application

An *anemometer* is a sensor that measures air velocity. Anemometers may be deployed in a building to diagnose problems with the Heating, Ventilation, and Cooling system (HVAC), and also to collect air flow measurements for improved HVAC control. This requires anemometers in difficult-to-reach locations, such as in air flow ducts, where it is infeasible to run wires. Therefore, anemometers must be battery-powered and must transmit readings wirelessly, making LLNs attractive.

We used anemometers based on the Hamilton platform [19], each consisting of four ultrasonic transceivers arranged as vertices of a tetrahedron (Figure 13). To measure the air velocity, each transceiver, in turn, emits a burst of ultrasound, and the impulse is measured by the other three transceivers. This process results in a total of 12 measurements.

Calculating the air velocity from these measurements is computationally infeasible on the anemometer itself, because Hamilton does not have hardware floating point support and the computations require complex trigonometry. Measurements must be transmitted over the network to a server that



(a) Anemometer (b) Hamilton-based PCB (bottom and top)

Figure 13: Hamilton-based ultrasonic anemometer

processes the data. Furthermore, a specific property of the analytics is that it requires a contiguous stream of data to maintain calibration (a numerical integration is performed on the measurements). Thus, the application requires a high sample rate (1 Hz), and is sensitive to data loss. A protocol for *reliable* delivery, like TCP or CoAP, is therefore necessary.

We note that the 1 Hz sample rate for this application is much higher than the sample rate of most sensors deployed in buildings. For example, a sensor measuring temperature, humidity, or occupancy in a building typically only generates a single reading every few tens of seconds or every few minutes. Furthermore, each individual reading from the anemometer is quite large (82 bytes), given that it encodes all 12 measurements (plus a small header). Given the higher data rate requirements of the anemometer application, we plan to use a higher-capacity battery than the standard AA batteries used in most motes. The higher cost of such a battery is justified by the higher cost of the anemometer transducers.